



那一年，我们写过的代码

Java程序员

——从笨鸟到菜鸟



CSDN博主 曹胜欢

- 总结了作者长期学习成果，实用性强
- 系统全面的介绍了Java的技术要点
- 精心编写了博客，引导性强，难易适中
- 网站博客提供代码下载和详细讲解支持



作者保留所有版权所有



作者简介：



曹胜欢：滨州学院计算机科学与技术系软件技术 2010 级学生，专科。在 2011 年正式开始接触软件开发，至今经历了头疼——迷惑——迷茫——开阔——漫长自学道路。在接触软件开发至今亲身经历了 IT 学习自学道路上的迷惑，所以从 2012 年 2 月份开始着手《java 程序员从笨鸟到菜鸟》的编写。真心希望可以帮助刚起步学习 java 开发的兄弟姐妹们。没参与过中大型项目的开发，没有高的学历。所以此人之书只能供参考。

写在前面

Java 作为当前 web 开发和 web 开发都有着众多开发者的流行语言，目前拥有着越来越多的学习者，本书面向的就是众多的初学者，作者亲身体验过初学时找不到方向的痛苦，所以从 2012 年 2 月开始有关本书的博客编写。经过半年时间的编写，终于完成大部分技术点的总结。如果你是大牛，在软件开发行业中已经滚打多年的老程序员，希望你能为书中提到不对的地方提出您宝贵的意见。也算是为中国软件开发行业做些应有的贡献。本书特点：对基础知识做了系统的讲解，但不是面面俱到，没有把每个知识点都讲解的很详细，本书主要是对 SSH 框架中做了深入详细的探讨，阅读本书的读者最好能有一定的基础，可以作为学习期间的参考书，或者是 SSH 框架的学习资料。

作为还未走出菜鸟的笼子的我，希望有更多的学妹学弟可以通过此书找到学习 java 的方向，少走一些我以前走过的弯路。由于本书只是为了大家传阅方便，只是以电子书的形式存在于网络，所以在排版上，只是做了一些简单的排版，排版有点不正规，并且目录和正文页码有些出入，我们就以目录为准。所以还请大家见谅。本电子书在一段时间后还会有更新。它将伴随着作者毕业。以后有新的技术博客更新，我会分批的更新本电子书。希望我能把我大学里所学的所以技术分享给大家，也希望有更多的朋友能够受益。也希望有更多的大牛能把自己的学习成果分享给大家，为中

国的软件事业做点贡献（貌似说的有点大了），总之，希望大家共同学习，共同进步

在本电子书的整理过程中，得到了许多网友、同学还有老师的支持，在此，真心感谢烟台南山学院李明娇同学的封面设计，滨州学院 2011 级王涛同学的网络编辑。滨州学院计算机系 2010 级金振成、陈西东和某软件公司争光等同等战线的兄弟们的技术支持。还要感谢滨州学院计算机系冯君、孙继磊、庄波等老师一直以来的指导和技术支持。在此对这些帮助过我的老师和同学们表示衷心的感谢

由于本书都是从作者博客中摘抄过来的，基本没有经过什么正规的校验，所以难免会出现一些不对的地方。欢迎各位大牛对本书提出批评。

作者博客：[**http://blog.csdn.net/csh624366188**](http://blog.csdn.net/csh624366188)

作者邮箱：www.bzu901@163.com

曹胜欢

2012 年 9 月 20 日凌晨

目录

作者简介:	2
写在前面.....	3
目录.....	5
(一)开发环境搭建,基本语法,字符串,数组	9
(二)面向对象之封装,继承,多态(上)	25
(三)面向对象之封装,继承,多态(下)	36
(四)java 开发常用类(包装,数字处理集合等)(上)	46
(五) java 开发常用类(包装,数字处理集合等)(下)	58
(六) I/O 流操作	74
(七) ——java 数据库操作	89
(八) 反射和代理机制	116
(九) ——数据库有关知识补充(事务、视图、索引、存储过程)	125
(十) 枚举,泛型详解	137
(十一) 多线程讲解	145
(十二) java 异常处理机制	155
(十三) java 网络通信编程	169
(十四) Html 基础积累总结(上)	186
(十五) Html 基础积累总结(下)	197
(十六) CSS 基础积累总结(上)	205
(十七) CSS 基础积累总结(下)	215
(十八) JSP 基本语法与动作指令	233
(十九)EL 表达式和 JSTL	241
(二十) jsp 自定义标签	252
(二十一) java 过滤器和监听器详解	259
(二十二) 华山论 session 和 cookie 机制	272
(二十三) 常见乱码解决以及 javaBean 基础知识	279
(二十四) Xml 基础详解和 DTD 验证	285
(二十五) XML 之 Schema 验证	293
(二十六) XML 之 DOM 和 SAX 解析	304
(二十七) XML 之 Jdom 和 DOM4J 解析	319
(二十八) Javascript 总结之语言基础	329
(二十九) javascript 对象的创建和继承实现	340
(三十) javascript 弹出框、事件、对象化编程	353
(三十一) 大话设计模式(一)设计模式遵循的七大原则	369
(三十二) 大话设计模式(二)设计模式分类和三种工厂模式	379
(三十三) 大话设计模式(三)单例模式	396
(三十四) 大话设计模式(四)策略模式	402
(三十五) 大话设计模式(五)创建者模式和原型模式	410

(三十六) 大话设计模式 (六) 观察者模式	431
(三十七) 大话设计模式 (七) 代理模式和 java 动态代理机制	480
(三十八) 大话设计模式 (八) 状态模式	515
(三十九) 大话设计模式 (九) 迭代器模式和命令模式	549
(四十) 细谈 struts2 (一) 自己实现 struts2 框架	424
(四十一) 细谈 struts2 (二) 开发第一个 struts2 的实例	453
(四十二) 细谈 struts2 (三) struts2 拦截器源码分析	460
(四十三) 细谈 struts2 (五) action 基础知识和数据校验	503
(四十四) 细谈 struts2 (六) 获取 servletAPI 和封装表单数据	524
(四十五) 细谈 struts2 (七) 数据类型转换详解	535
(四十六) 细谈 struts2 (八) 拦截器的实现原理及源码剖析	561
(四十七) 细谈 struts2 (九) 内置拦截器和自定义拦截器详解(附源码)	572
(四十八) 细谈 struts2 (十) ognl 概念和原理详解	584
(五十) 细谈 struts2 (十一) OGNL 表达式的基本语法和用法	591
(五十一) 细谈 struts2 (十二) struts2 国际化底层大揭秘	773
(五十二) 细谈 struts2 (十三) struts2 实现文件上传和下载详解	820
(五十三) 细谈 struts2 (十四) struts2+ajax 实现异步验证	855
(五十四) 细谈 Hibernate (一) hibernate 基本概念和体系结构	601
(五十五) 细谈 Hibernate (二) 开发第一个 hibernate 基本详解	605
(五十六) 细谈 Hibernate (三) Hibernate 常用 API 详解及源码分析	623
(五十七) 细谈 Hibernate (四) Hibernate 常用配置文件详解	633
(五十八) 细谈 Hibernate (五) Hibernate 一对多关系映射	653
(五十九) 细谈 Hibernate (六) Hibernate 继承关系映射	664
(六十) 细谈 Hibernate (七) Hibernate 自身一对多和多对多关系映射	682
(六十一) 细谈 Hibernate (八) Hibernate 集合 Map 关系映射	696
(六十二) 细谈 Hibernate (九) hibernate 一对一关系映射	705
(六十三) 细谈 Hibernate (十) hibernate 查询排序和组件映射	716
(六十四) 细谈 Hibernate (十一) hibernate 复合主键映射	725
(六十五) 细谈 Hibernate (十二) hibernate 查询排序组件映射	734
(六十六) 细谈 Hibernate (十三) session 缓存机制和三种对象状态	741
(六十七) 细谈 Hibernate (十四) Hibernate 三种检索方式详解	751
(六十八) 细谈 Hibernate (十五) HQL 与 QBC 查询方式详解	756
(六十九) 细谈 Hibernate (十六) 数据库事务与隔离级别	766
(七十) 细谈 Hibernate (十七) Hibernate 实现分页和综合查询详解	799
(七十一) 细谈 Hibernate (十八) 悲观锁和乐观锁解决 hibernate 并发	861
(七十二) 细谈 Hibernate (十九) Hibernate 二级缓存详解	869
(六十三) 细谈 Spring (一) spring 简介	785
(六十四) 细谈 Spring (二) 自己动手模拟 spring	793
(七十五) 细谈 Spring (三) IOC 和 spring 基本配置详解	805
(七十六) 细谈 Spring (四) 利用注解实现 spring 基本配置详解	829
(七十七) 细谈 Spring (五) spring 之 AOP 底层大揭秘	838
(七十八) 细谈 Spring (六) spring 之 AOP 基本概念和配置详解	845
(七十九) 细谈 Spring (七) spring 之 JDBC 访问数据库及配置详解	877
(八十) 细谈 Spring (八) spring+hibernate 整合基本详解	881

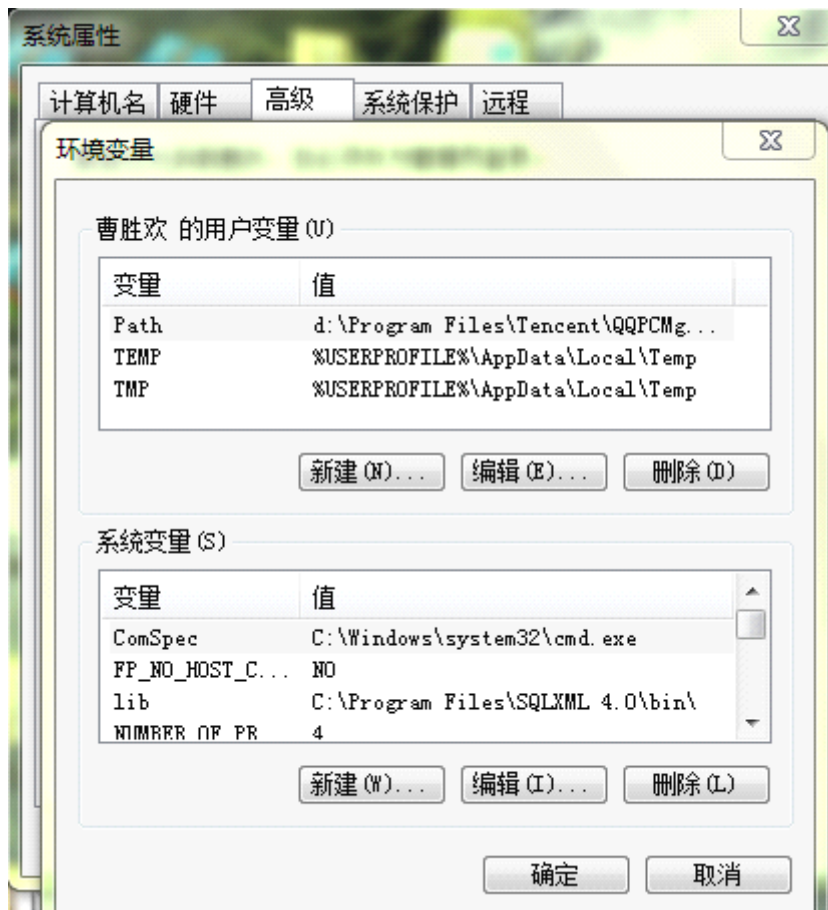
(八十一) 细谈 Spring (九) spring+hibernate 声明式事务管理详解	888
(八十一) 细谈 Spring(十)深入源码分析 Spring 之 HibernateTemplate 和 HibernateDaoSupport	902
(八十二) 细谈 Spring (十一) 深入理解 spring+struts2 整合 (附源码)	910
(八十三) 细谈 Spring (十二) OpenSessionInView 详解及用法	920
(八十四) 深入浅出 Ajax	925
(八十五) 跟我学 jquery (一) 爱之初体验 jquery	932
(八十六) 跟我学 jquery (二) 大话 jquery 选择器	943
(八十七) 跟我学 jquery (三) jquery 动态创建元素和常用函数示例	961
(八十八) 跟我学 jquery (四) JQuery 框架操作元素的属性与样式	975
(八十九) 跟我学 jquery (五) jquery 中的 ajax 详解	994
(九十) 跟我学 jquery (六) jquery 中事件详解	1012
(九十一) 跟我学 jquery (七) jquery 动画大体验	1033
(九十二) 深入 java 虚拟机 (一) ——java 虚拟机底层结构详解	1045
(九十三) 深入 java 虚拟机 (二) ——类加载器详解 (上)	1053
(九十四) 深入 java 虚拟机 (三) ——类加载器(中) 类的初始化	1062
附录.....	1066
一: 基于 SSH 实现的简单学生选课系统 (附源码)	1066
二: 基于 SSH 的商场管理系统(附源码)	1080
三: 北京实习总结——记住牛人那些话.....	1086

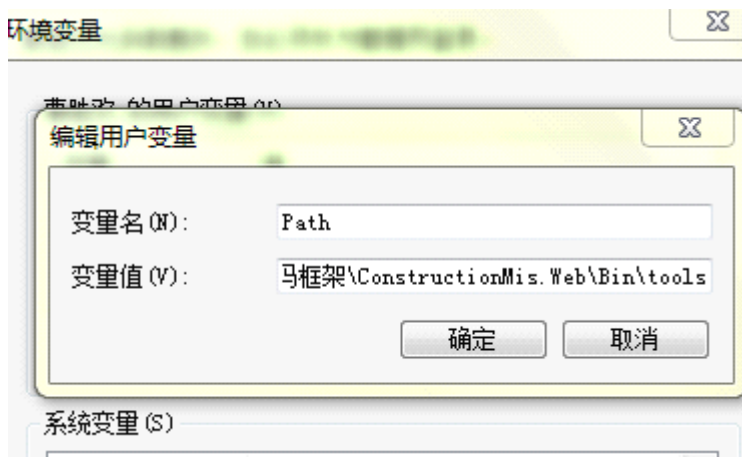
(一)开发环境搭建，基本语法，字符串，数组

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

今天进行第一块的复习，首先是环境的搭建，java 开发的首先任务就是环境变量的配置和环境的搭建，虽然现在大多数的开发工具都已经集成了这些，但是对于一个初学者来说，了解这些还是有些必要的，首先先配置一下JDK，下载JDK在sun公司的官网里下载即可（虽然被oracle收购了，但还是习惯叫他sun公司），找到jdk安装路径，然后把路径黏贴到环境变量path里面，有图有真相，上图：





然后在 dos 命令里面测试一下，如果有下面结果即为配置成功：

```
管理员: C:\Windows\system32\cmd.exe

-classpath <class search path of directories and zip/jar files>
           A ; separated list of directories, JAR archives,
           and ZIP archives to search for class files.
-D<name>=<value>
           set a system property
-verbose[:class!gc!jnil
           enable verbose output
-version   print product version and exit
-version:<value>
           require the specified version to run
-showversion print product version and continue
-jre-restrict-search ! -jre-no-restrict-search
           include/exclude user private JREs in the version search
-? -help   print this help message
-X         print help on non-standard options
-ea[:<packagename>...!:<classname>]
-enableassertions[:<packagename>...!:<classname>]
           enable assertions
-da[:<packagename>...!:<classname>]
-disableassertions[:<packagename>...!:<classname>]
           disable assertions
-esa ! -enablesystemassertions
           enable system assertions
-dsa ! -disablesystemassertions
           disable system assertions
```

环境变量配置好之后，我们就可以进行我们的 java 开发之旅了，下面进行 java 基本语法的复习了：

一：首先说一下 java 中的语句规则：

1.java 每行代码以；结尾。

2.Java 中的注释有：

(1) // 注释一行

以“//”开始，终止于行尾，一般作单行注释，可放在语句的后面

(2) /*.....*/ 一行或多行注释

以“/*”开始，最后以“*/”结束，中间可写多行。

(3) /**.....*/

以“/**”开始，最后以“*/”结束，中间可写多行。这种注释主要是为支持 JDK 工具 javadoc 而采用的。

3.Java 中的合法标示符要符合一下规则： 1

1) 要以大小写字母或者美元符号或者下划线开头，不能以数字开头

2) 标示符命名不能用关键字，关键字是 java 内部所用到的标示符，为了避免混淆，所以不能用。

3) 类，变量，方法名命名尽量有一定规则，不要随便命名，虽然系统不会报错，但是为了项目开发后期的维护，所以尽量起比较有意义的名字，并且命名要符合一定的规则，如驼峰规则。

二：java 基本数据类型

Java 数据类型被分为：基本数据类型和引用数据类型。

Java 中有 8 中基本数据类型：

类型	位长/b	默认值	取值范围
布尔型 (boolean)	1	false	true false
字节型 (byte)	8	0	-128~127
字符型 (char)	16	'\u0000'	'\u0000'~'\uffff'即 0~65535
短整型(short)	16	0	-32768~32767

整型(int)	32	0	-231~231-1
长整型 (long)	64	0	-263~263-1
单精度 (float)	32	0.0	+ -1.4E-45 或 + -3.4028235E+38
双精度 (double)	64	0.0	+ -4.9E-324 或 + -1.797693134862315E+308

注：String 类型不是基本数据类型，它被定义为类，属于引用数据类型。，
由于字符串是常用的数据类型。Java 提供了对 String 类型特殊操作，直接引用，例如：String s="hello world";

三：引用类型

引用类型是一个对象类型的，它的值是指向内存空间的引用，就是地址，所指向的内存中保存着变量所表示的一个值或一组值。很好理解吧，因为一个对象，比如说一个人，不可能是个数字也不可能是个字符啊，所以要想找它的话只能找它的地址了。

接下来看看基本类型和引用类型变量的不同处理吧。基本类型自然是简单，声明是自然系统就给它空间了。例如，

```
int baijq;
```

```
baijq=250; //声明变量 baijq 的同时，系统给 baijq 分配了空间。
```

引用类型就不是了，只给变量分配了引用空间，数据空间没有分配，因为谁都不知道数据是什么啊，整数，字符？我们看一个错误的例子：

```
MyDate today;
```

```
today.day = 4; //发生错误，因为 today 对象的数据空间未分配。
```

那我们怎么给它赋值啊？引用类型变量在声明后必须通过实例化开辟数据空间，才能对变量所指向的对象进行访问。举个例子：

```
MyDate today;           //将变量分配一个保存引用的空间
```

```
today = new MyDate ();   //这句话是 2 步，首先执行 new MyDate (),  
给 today 变量开辟数据空间，然后再执行赋值操作。
```

四：定义变/常量和变量的初始化

Java 定义变量结构：类型 变量名，这里的变量名要符合标示符规则

1. 变量的声明

格式：类型 变量名[, 变量名]=初值, ... ;

赋值：[类型] 变量名=值

如：int a=5 , b=6 , c , d ;

说明：

(1) 变量名必须在作用域中是唯一的，不同作用域中才允许相同名字的变量出现；

(2) 只要在同一代码块没有同名的变量名，可以在程序中任何地方定义变量，一个代码块就是两个相对的“{ }”之间部分。

2. 变量的使用范围

每个变量的使用范围只在定义它的代码块中，包括这个代码块中包含的代码块。

在类开始处声明的变量是成员变量，作用范围在整个类；

在方法和块中声明的变量是局部变量，作用范围到它的“}”；

3. 变量类型的转换

Java 可以将低精度的数字赋值给高精度的数字型变量，反之则需要强制类型转换。

强制转换格式：（数据类型）数据表达式

字节型 短整型 字符型 整型 长整型 单精度实型 双精度实型

转化规律：由低到高

变量与存储器有着直接关系，定义一个变量就是要编译器分配所需要的内存空间，分配多少空间，这就是根据我们所定义的变量类型所决定的。变量名实际上是代表所分配空间的内存首地址

常量

Java 中的常量值是用文字串表示的，它区分为不同的类型，如整型常量 123，实型常 1.23，

字符常量'a'，布尔常量 true、false 以及字符串常量“This is a constant string”。

Java 的常量用 final 说明，约定常量名一般全部使用大写字母，如果是多个单词组合在一起的，单词之间用下划线连接，常量在程序执行时不可更改。

如：final int i=1;

i=i+1; //错，i 是 final（常量），不可更改值

例如：final double IP = 3.14159 D

说明：默认类型为 64 位 double 双精度类型(D 或 d),数字后面加 F 或 f 则是 32 位 float 单

精度(实数)类型

五：运算符

1、赋值运算符

赋值运算符用于把一个数赋予一个变量。赋值运算符两边的类型不一致时，那么如果左侧的数据类型的级别高，则右边的数据将转换成左边的数据类型在赋予左边的变量，否则需要强制类型转换。

赋值运算符包括=、+=、-=、*=、%=、/=等。

2、算术运算符

算术运算符用于对整型数或者浮点数进行运算，java 语言中的算术运算符包括二元运算符和一元运算符。所谓的几元运算符即参加运算的操作数的个数。

1) 二元运算符

Java 的二元运算符有+（加）、-（减）、*（乘）、/（除）、%（取余数）。

2) 一元运算符

Java 的一元运算符有++（自加）、--（自减）

3、关系运算符

关系运算符用来比较两个值，返回布尔类型的值 true 或 false。

等于 不等于 小于 小于等于 大于等于 大于

== != < <= >= >

4、条件运算符

条件运算符的作用是根据表达式的真假决定变量的值。

1> 格式：条件表达式 ? 表达式 2 : 表达式 3

2> 功能：条件表达式为 true，取“表达式 2”值，为 false，取“表达式 3”的值

例： int a=0x10 , b=010 , max ;

max=a>b ? a : b ;

```
System.out.println(max); // 输出 16
```

5、逻辑运算符

运算符 结果

~ 按位非（NOT）（一元运算）

& 按位与（AND）

| 按位或（OR）

^ 按位异或（XOR）

>> 右移

>>> 右移，左边空出的位以 0 填充；无符号右移

<< 左移

&= 按位与赋值

|= 按位或赋值

^= 按位异或赋值

>>= 右移赋值

>>>= 右移赋值，左边空出的位以 0 填充；无符号左移

<<= 左移赋值

按位非（NOT）

按位非也叫做补，一元运算符 NOT“~”是对其运算数的每一位取反。例如，

数字 42，它的二进制代码为： 00101010

经过按位非运算成为 11010101

六：流程控制语句

分支语句

1.简单的 if....else 语句

```
If(条件){
```

如果条件为真、、、、

```
}
```

```
Else{
```

如果条件为假、、、、、、

```
}
```

2、只有 if 的语句：

```
If(条件) {
```

如果条件为真，执行。。。如果为假，不执行

```
}
```

3、switch 语句是多分枝语句，基本语法：

```
Switch(expr){
```

```
Case value1:
```

```
Statements;
```

```
Break;
```

```
.....
```

```
Case valueN:
```

```
Statements;
```

```
Break;
```

```
Default:
```

```
Statements;
```

```
Break;
```

```
}
```

注：1.expr 必须是与 int 类型兼容的类型，即为 byte, short, char 和 int 类型中的其中一种

2.Case valueN:中 valueN 也必须是 int 类型兼容的类型，并且必须是常量

3.各个 case 子句的 valueN 表达式的值不同

4.Switch 语句中只能有一个 default 子句。

循环语句

1.while 语句 2.do.....while 语句（此处省略三百字）

3.for 语句

基本格式：for(初始化；循环条件；迭代部分)

功能：（1）第一次进入 for 循环时，对循环控制变量赋初值；

（2）根据判断条件检查是否要继续执行循环。为真执行循环体内语句块，为假则结束循环；

（3）执行完循环体内语句后，系统根据“循环控制变量增减方式”改变控制变量值，再回

（3）到步骤（2）根据判断条件检查是否要继续执行循环。

4.流程跳转语句：break,continue 和 return 语句用来控制流程的跳转

1) break: 从 switch 语句，循环语句或标号标识的代码块中退出

2) continue: 跳出本次循环，执行下次循环，或执行标号标识的循环体；

3) return: 退出本方法，跳到上层调用方法。

4) Break 语句和 continue 语句可以与标号联合使用。标号用来标识程序中的语句，标号的名字可以是任意的合法标识符。

带有标号的循环体：

```
Loop: switch(expr){  
}
```

七：字符串

字符串的几种用法：

拼接 直接用“+”把两个字符串拼接起来

例如: `String firstName = "li";`

`String secondName = "ming";`

`String fullName = firstName+secondName;`

检测字符串是否相等 检测两个字符串内容是否相等时使用“`equals`”; 比较两个字符串的引用是否相等时用“`==`”

得到字符串的长度 字符串变量名.`length()`;

String, StringBuffer, StringBuilder 区别

String 字符串常量

StringBuffer 字符串变量（线程安全）

StringBuilder 字符串变量（非线程安全）

简要的说, `String` 类型和 `StringBuffer` 类型的主要性能区别其实在于 `String` 是不可变的对象, 因此在每次对 `String` 类型进行改变的时候其实都等同于生成了一个新的 `String` 对象, 然后将指针指向新的 `String` 对象, 所以经常改变内容的字符串最好不要用 `String`, 因为每次生成对象都会对系统性能产生影响, 特别当内存中无引用对象多了以后, `JVM` 的 `GC` 就会开始工作, 那速度是一定会相当慢的。

而如果是使用 `StringBuffer` 类则结果就不一样了, 每次结果都会对 `StringBuffer` 对象本身进行操作, 而不是生成新的对象, 再改变对象引用。所以在一般情况下我们推荐使用 `StringBuffer`, 特别是字符串对象经常改变的情况下。而在某些特别情况下, `String` 对象的字符串拼接其实是被 `JVM` 解释成了 `StringBuffer` 对象的拼接, 所以这些时候 `String` 对象的速度并不会比 `StringBuffer` 对象慢, 而特别是以下的字符串对象生成

中， `String` 效率是远要比 `StringBuffer` 快的：

```
String S1 = "This is only a" + " simple" + " test";  
StringBuffer Sb = new StringBuilder("This is only a").append(" simple").append(" test");
```

你会很惊讶的发现，生成 `String S1` 对象的速度简直太快了，而这个时候 `StringBuffer` 居然速度上根本一点都不占优势。其实这是 `JVM` 的一个把戏，在 `JVM` 眼里，这个

`String S1 = "This is only a" + " simple" + "test";` 其实就是：

`String S1 = "This is only a simple test";` 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 `String` 对象的话，速度就没那么快了，譬如：

```
String S2 = "This is only a";  
String S3 = " simple";  
String S4 = " test";  
String S1 = S2 +S3 + S4;
```

这时候 `JVM` 会规规矩矩的按照原来的方式去做

在大部分情况下 `StringBuffer > String`

`StringBuffer`

`Java.lang.StringBuffer` 线程安全的可变字符序列。一个类似于 `String` 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。

可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。

`StringBuffer` 上的主要操作是 `append` 和 `insert` 方法，可重载这些方法，以

接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。 `append` 方法始终将这些字符添加到缓冲区的末端；而 `insert` 方法则在指定的点添加字符。

例如，如果 `z` 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 `z.append("le")` 会使字符串缓冲区包含“startle”，而 `z.insert(4, "le")` 将更改字符串缓冲区，使之包含“starlet”。

在大部分情况下 `StringBuilder > StringBuffer` `java.lang.StringBuilder`

`java.lang.StringBuilder` 一个可变的字符序列是 5.0 新增的。此类提供一个与 `StringBuffer` 兼容的 API，但不保证同步。该类被设计用作 `StringBuffer` 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 `StringBuffer` 要快。两者的方法基本相同。

八、数组

数组是有序数据的集合，数组中的每个元素具有相同的数组名，根据数组名和下标来唯一确定数组中的元素。使用时要先声明后创建

1、一位数组

1) 一维数组的声明

格式：数据类型 数组名[] 或 数据类型 []数组名

例： `int a[] ; String s[] ; char []c ;`

说明：定义数组，并不为数据元素分配内存，因此“[]”中不用指出数组中元素个数。

2) 一维数组的创建与赋值

创建数组并不是定义数组，而是在数组定义后，为数组分配存储空间，同时对数组元素进行初始化

(1) 用运算符 **new** 分配内存再赋值

格式：数组名=new 数据类型[size]

例：int a[]；

a=new int[3]；// 产生 a[0]，a[1]，a[2] 三个元素

a[0]=8；a[1]=8；a[2]=8；

3) 直接赋初值并定义数组的大小

例：int i[]={4,5,0,10,7,3,2,9}；

String names[]={“张三”,“李四”,“王五”,“宋七”}；

4) 测试数组长度（补充）

格式：数组名.length

char c[]={‘a’,‘b’,‘c’,‘北’,‘京’}；

System.out.print(c.length)； // 输出 5

2、多维数组

以二维数组为例

例：int d[][]；// 定义一个二维数组

d=new int[3][4]；// 分配 3 行 4 列数组内存

int a[][]=new int[2][]；

a[0]=new int[3]；// 第二维第一个元素指向 3 个整型数

a[1]=new int[5]；// 第二维第一个元素指向 5 个整型数

注意：Java 可以第二维不等长

int i[][]={{0},{1,4,5},{75,6},{8,50,4,7}}；//定义和赋初值在一起

下面数组定义正误的判断

```
int a[][]=new int[10,10] //错
```

```
int a[10][10]=new int[][] //错
```

```
int a[][]=new int[10][10] //对
```

```
int []a[]=new int[10][10] //对
```

```
int [][]a=new int[10][10] //对
```

注意：java 中二维数组分配空间是第二维可以为空，但是第一维必须分配内存。

最后以一个经典的 **helloworld** 代码结束本章的总结

```
public class HelloWorldApp{  
public static void main(String[] args){  
System.out.println("hello world!");  
}  
}
```

(二)面向对象之封装，继承，多态（上）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

Java 是一种面向对象的语言，这是大家都知道的，他与那些像 c 语言等面向对象语言不同的是它本身所具有的面向对象的特性——封装，继承，多态，这也就是传说中的面向对象三大特性

一：从类和对象开始说起：

Oop: Object Oriented Programming(面向对象编程):

类：对象的蓝图，生成对象的模板，是对一类事物的描述，是抽象的概念上的定义

对象：对象是实际存在的该类事物的每个个体，因而也称为实例

类之间的三种关系：依赖关系(uses-a)聚集关系(has-a)继承关系(is-a)
在 java 中，类和对象的关系就像是动物和老虎的关系一样，老虎属于动物，老虎只是动物的一个实例。

类表示了对象的类别，是创建对象的蓝本。建立一个事物的抽象模型，本质上就是表达该事物的性质和行为。使用类来建立抽象模型，是通过在类中定义变量和方法来实现的。

类中定义的属性是一种可存储值的变量，该变量代表着事物的具体性质。类的对象所具有的行是由定义的方法来实现的。类中定义的变量和方法被称为类的成员。

对象是类的实例。对象在构造时以类为蓝本，创建对象的过程叫做实例化。

对象是类所表示的抽象事物的具体实例。

面向对象：

- 1：将复杂的事情简单化。
- 2：面向对象将以前的过程中的执行者，变成了指挥者。
- 3：面向对象这种思想是符合现在人们思考习惯的一种思想。

过程和对象在我们的程序中是如何体现的呢？

过程其实就是函数。

对象是将函数等一些内容进行了封装。

匿名对象使用场景：

- 1：当对方法只进行一次调用的时候，可以使用匿名对象。
- 2：当对象对成员进行多次调用时，不能使用匿名对象。必须给对象起名字。

在类中定义其实都称之为成员。成员有两种：

- 1：成员变量：其实对应的就是事物的属性。
- 2：成员函数：其实对应的就是事物的行为。

所以，其实定义类，就是在定义成员变量和成员函数。但是在定义前，必须先要对事物进行属性和行为的分析。才可以用代码来体现。

`private int age;`//私有的访问权限最低，只有在本类中的访问有效。

注意：私有仅仅是封装的一种体现形式而已。

私有的成员：其他类不能直接创建对象访问，所以只有通过本类对外提供具体的访问方式来完成对私有的访问。

可以通过对外提供函数的形式对其进行访问。

好处：可以在函数中加入逻辑判断等操作，对数据进行判断等操作。

总结：开发时，记住，属性是用于存储数据的。直接被访问，容易出现安全隐患。所以，类中的属性通常被私有化，并对外提供公共的访问方法。

这个方法一般有两个，规范写法：对于属性 `xxx`，可以使用 `setXXX()`,`getXXX()` 对其进行操作。

类中怎么没有定义主函数呢？

注意：主函数的存在，仅为该类是否需要独立运行。如果不需要，主函数是不用定义的。

成员变量和局部变量的区别：

1：成员变量直接定义在类中。

局部变量定义在方法中，参数上，语句中。

2：成员变量在这个类中有效。

局部变量只在自己所属的大括号内有效，大括号结束，局部变量失去作用域。

3：成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。

局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

构造函数：用于给对象进行初始化，是给与之对应的对象进行初始化，它具有针对性，函数中的一种。

特点：

1：该函数的名称和所在类的名称相同。

2：不需要定义返回值类型。

3：该函数没有具体的返回值。

记住：所有对象创建时，都需要初始化才可以使用。

注意事项：

一个类在定义时，如果没有定义过构造函数。那么该类中会自动生成一个空参数的构造函数。为了方便该类创建对象，完成初始化。

如果在类中自定义了构造函数。那么默认的构造函数就没有了。

简单一句话：你写了，默认就没有了。你没写就只有默认的。

一个类中，可以有多个构造函数，因为它们的函数名称都相同，所以只能通过参数列表来区分。所以，一个类中如果出现多个构造函数。它们的存在是以重载体现的。

构造函数和一般函数有什么区别呢？

- 1：两个函数定义格式不同。
- 2：构造函数是在对象创建时，就被调用，用于初始化，而且初始化动作只执行一次。

一般函数，是对象创建后，需要调用才执行，可以被调用多次。

什么时候使用构造函数呢？

分析事物时，发现具体事物一出现，就具备了一些特征，那就将这些特征定义到构造函数内。

构造代码块和构造函数有什么区别？

构造代码块：是给所有的对象进行初始化，也就是说，所有的对象都会调用一个代码块。只要对象一建立。就会调用这个代码块。

构造函数：是给与之对应的对象进行初始化。它具有针对性。

```
Person p = new Person();
```

创建一个对象都在内存中做了什么事情？

- 1: 先将硬盘上指定位置的 **Person.class** 文件加载进内存。
- 2: 执行 **main** 方法时，在栈内存中开辟了 **main** 方法的空间(压栈-进栈)，然后在 **main** 方法的栈区分配了一个变量 **p**。
- 3: 在堆内存中开辟一个实体空间，分配了一个内存首地址值。**new**
- 4: 在该实体空间中进行属性的空间分配，并进行了默认初始化。
- 5: 对空间中的属性进行显示初始化。
- 6: 进行实体的构造代码块初始化。
- 7: 调用该实体对应的构造函数，进行构造函数初始化。（）
- 8: 将首地址赋值给 **p**，**p** 变量就引用了该实体。(指向了该对象)

二：类的访问权限

(1)public:

对于成员来说：任何其他类都可以访问它们，不管在同一个包中还是在另外的包中。

对于类来说： 也是一样。

(2)friendly:

对于成员来说：如果一个类的成员没有任何权限修饰，那么它门就是缺省包访问权限，用 **friendly** 来表示，注意 **friendly** 不是 Java 中的关键字，这里是个人喜欢的方式用它表示而已。同一个包内其它类可以访问，但包外就不可以。对于同一个文件夹下的、没有用 **package** 的 **classes**，Java 会自动将这些 **classes** 初见为隶属于该目录

的 default package，可以相互调用 class 中的 friendly 成员。如以下两个 class 分别在同一个文件夹的两个文件

中，虽然没有引入 package，但隶属于相同的 default package。

对于类来说：同一个包中的类可以用。总之，类只可以声明为 public 或者 friendly。

(3)private:

对于对于成员来说：只能在该成员隶属于的类中访问。

对于类来说：类不可以声明为 private。

4)protected:

对于对于成员来说：相同包中的类可以访问(包访问权限);基类通过 protected 把基类中的成员的访问权限赋予派生类不是所有类（派生类访问权限）。

对于类来说：类不可以声明为 protected

说明：

- 1、每个编译单元（类文件）都仅能有一个 public class
- 2、public class 的名称（包含大小写）必须和其类文件同名。
- 3、一个类文件(*.java)中可以不存在 public class。

这种形式的存在的场景：如果我们在某个包内撰写一个 class，仅仅是为了配合同包内的其他类工作，而且

我们不想再为了撰写说明文档给客户（不一定是现实意义的客户，可能是调用这个类的类）看而伤脑筋，而且有可能过一段时间之后

有可能会彻底改变原有的做法，并完全舍弃旧版本，以全新的版本代替。

4、class 不可以是 private 和 protected。

5、如果不希望那个任何产生某个 `class` 的对象，可以将该类得所有构造函数设置成 `private`。但是即使这样也可以生成该类的对象，就是 `class` 的 `static` 的成员（属性和方法）可以办到。

三、面向对象之——封装

封装：顾名思义，隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别；将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成“类”，其中数据和函数都是类的成员。

封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过 外部接口，一特定的访问权限来使用类的成员。

封装的大致原则:

- 1 把尽可能多的东西藏起来.对外提供简捷的接口.
- 2、把所有的属性藏起来.
- 3、封装好处：将变化隔离；便于使用；提高重用性；安全性。

四： **this** 和 **static** 详解

this:代表对象。就是所在函数所属对象的引用。

this 到底代表什么呢？哪个对象调用了 **this** 所在的函数。**this** 就代表哪个对象。就是哪个对象的引用。

开发时，什么时候使用 **this** 呢？

在定义功能时，如果该功能内部使用到了调用该功能的对象。这时就用 **this** 来表示这个对象。

this 还可以用于构造函数间的调用。

调用格式：**this**(实际参数);

this 对象后面跟上 . 调用的是成员属性和成员方法(一般方法);

this 对象后面跟上 () 调用的是本类中的对应参数的构造函数。

注意：用 **this** 调用构造函数，必须定义在构造函数的第一行。因为构造函数是用于初始化的，所以初始化动作一定要执行。否则编译失败。

static：关键字，是一个修饰符。用于修饰成员(成员变量和成员函数)。

特点：

- 1，想要实现对象中的共性数据的对象共享。可以将这个数据进行静态修饰。
- 2，被静态修饰的成员，可以直接被类名所调用。也就是说，静态的成员多了一种调用方式。类名.静态方式。
- 3，静态随着类的加载而加载。而且优先于对象存在。

弊端：

- 1，有些数据是对象特有的数据，是不可以被静态修饰的。因为那样的话，特有数据会变成对象的共享数据。这样对事物的描述就出了问题。所以，在定义静态时，必须要明确，这个数据是否是被对象所共享的。
- 2，静态方法只能访问静态成员，不可以访问非静态成员。
因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。
- 3，静态方法中不能使用 **this**，**super** 关键字。
因为 **this** 代表对象，而静态在时，有可能没有对象，所以 **this** 无法使用。
- 4，主函数是静态的。

什么时候定义静态成员呢？或者说：定义成员时，到底需不需要被静态修饰呢？

成员分两种：

1，成员变量。（数据共享时静态化）

该成员变量的数据是否是所有对象都一样：

如果是，那么该变量需要被静态修饰，因为是共享的数据。

如果不是，那么就说这是对象的特有数据，要存储到对象中。

2，成员函数。（方法中没有调用特有数据时就定义成静态）

如果判断成员函数是否需要被静态修饰呢？

只要参考，该函数内是否访问了对象中的特有数据：

如果有访问特有数据，那方法不能被静态修饰。

如果没有访问过特有数据，那么这个方法需要被静态修饰。

成员变量和静态变量的区别：

1，成员变量所属于对象。所以也称为实例变量。

静态变量所属于类。所以也称为类变量。

2，成员变量存在于堆内存中。

静态变量存在于方法区中。

3，成员变量随着对象创建而存在。随着对象被回收而消失。

静态变量随着类的加载而存在。随着类的消失而消失。

4，成员变量只能被对象所调用。

静态变量可以被对象调用，也可以被类名调用。

所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。

静态的注意：静态的生命周期很长。

静态代码块：就是一个有静态关键字标示的一个代码块区域。定义在类中。

作用：可以完成类的初始化。

静态代码块随着类的加载而执行，而且只执行一次（**new** 多个对象就只执行一次）。如果和主函数在同一类中，优先于主函数执行。

主函数的解释：

保证所在类的独立运行。是程序的入口。被 **jvm** 调用。

public:访问权限最大。

static: 不需要对象。直接类名即可。

void: 主函数没有返回值。

main: 主函数特定的名称。

(String[] args):主函数的参数，是一个字符串数组类型的参数。**jvm** 调用 **main** 方法时，传递的实际参数是 **new String[0]**。

jvm 默认传递的是长度为 0 的字符串数组。我们在运行该类时，也可以指定具体的参数进行传递。可以在控制台，运行该类时，在后面加入参数。参数

之间通过空格隔开。jvm 会自动将这些字符串参数作为 **args** 数组中的元素，进行存储。

静态代码块、构造代码块、构造函数同时存在时的执行顺序：

静态代码块 --构造代码块 --构造函数；

(三)面向对象之封装，继承，多态（下）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

上接：Java 程序员从笨鸟到菜鸟之（二）面向对象之封装，继承，多态（上）

五：再谈继承

继承是一种联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。私有成员能继承，但是由于访问权限的控制，在子类中不能直接使用父类的私有成员。并且 **java** 中是单继承，一个子类只能有一个父类

继承中的构造方法

当生成子类对象时，**Java** 默认首先调用父类的不带参数的构造方法，然后执行该构造方法，生成父类的对象。接下来，再去调用子类的构造方法，生成子类的对象。【要想生成子类的对象，首先需要生成父类的对象，没有父类对象就没有子类对象。比如说：没有父亲，就没有孩子】。

如果子类使用 **super()**显式调用父类的某个构造方法，那么在执行的时候就会寻找与 **super()**所对应的构造方法而不会再去寻找父类的不带参数的构

造方法。与 **this** 一样，**super** 也必须作为构造方法的第一条执行语句，前面不能有其他可执行语句。

当两个方法形成重写关系时，可以在子类方法中通过 **super.run()** 形式调用父类的 **run()** 方法，其中 **super.run()** 不必放在第一行语句，因此此时父类对象已经构造完毕，先调用父类的 **run()** 方法还是先调用子类的 **run()** 方法是根据程序的逻辑决定的。

方法的覆盖（重写）

重写的要求：子类覆盖方法和父类被覆盖方法的方法返回类型，方法名称，参数列表必须相同

子类覆盖方法的访问权限必须大于等于父类的方法的访问权限

方法覆盖只能存在于子类和父类之间

子类覆盖方法不能比父类被覆盖方法抛出更多异常

方法重写与方法重载之间的关系：重载发生在同一个类内部的两个或多个方法。重写发生在父类与子类之间。

final 关键字在继承中的使用

final 可以用于以下四个地方：

定义变量，包括静态的和非静态的。

如果 **final** 修饰的是一个基本类型，就表示这个变量被赋予的值是不可变的，即它是个常量；如果 **final** 修饰的是一个对象，就表示这个变量被赋予的引用是不可变的，不可改变的只是这个变量所保存的引用，并不是这个引用所指向的对象，其实更贴切的表述 **final** 的含义的描述，那就是，如果一个变量或

方法参数被 **final** 修饰，就表示它只能被赋值一次，但是 **JAVA** 虚拟机为变量设定的默认值不记作一次赋值。

被 **final** 修饰的变量必须被初始化。初始化的方式有以下几种：

1. 在定义的时候初始化。
2. 在初始化块中初始化。
3. 在类的构造器中初始化。
4. 静态变量也可以在静态初始化块中初始化。

1) 定义方法。

当 **final** 用来定义一个方法时，它表示这个方法不可以被子类重写，但是它这不影响它被子类继承。

说明：

具有 **private** 访问权限的方法也可以增加 **final** 修饰，但是由于子类无法继承 **private** 方法，因此也无法重写它。编译器在处理 **private** 方法时，是按照 **final** 方法来对待的，这样可以提高该方法被调用时的效率。不过子类仍然可以定义同父类中的 **private** 方法具有同样结构的方法，但是这并不会产生重写的效果，而且它们之间也不存在必然联系。

3) 定义类。

由于 **final** 类不允许被继承，编译器在处理时把它的所有方法都当作 **final** 的，因此 **final** 类比普通类拥有更高的效率。**final** 的类的所有方法都不能被重写，但这并不表示 **final** 的类的属性（变量）值也是不可改变的，要想做到 **final** 类的属性值不可改变，必须给它增加 **final** 修饰。

关于继承的几点注意：

- a) 父类有的，子类也有
- b) 父类没有的，子类可以增加
- c) 父类有的，子类可以改变
- d) 构造方法不能被继承
- e) 方法和属性可以被继承
- f) 子类的构造方法隐式地调用父类的不带参数的构造方法
- g) 当父类没有不带参数的构造方法时，子类需要使用 **super** 来显式地调用父类的构造方法，**super** 指的是对父类的引用
- h) **super** 关键字必须是构造方法中的第一行语句。

六：然后议多态

多态（Polymorphism）：用我们通俗易懂的话来说就是子类就是父类（猫是动物，学生也是人），因此多态的意思就是：**父类型的引用可以指向子类的对象。**

方法的重写、重载与动态连接构成多态性。**Java** 之所以引入多态的概念，原因之一是它在类的继承问题上和 **C++** 不同，后者允许多继承，这确实给其带来的非常强大的功能，但是复杂的继承关系也给 **C++** 开发者带来了更大的麻烦，为了规避风险，**Java** 只允许单继承，派生类与基类间有 IS-A 的关系（即“猫”is a “动物”）。这样做虽然保证了继承关系的简单明了，但是势必在功能上有很大的限制，所以，**Java** 引入了多态性的概念以弥补这点的不足，此外，抽象类和接口也是解决单继承规定限制的重要手段。同时，多态也是面向对象编程的精髓所在。

在一个类中，可以定义多个同名的方法，只要确定它们的参数个数和类型不同，这种现象称为类的多态。类的多态性体现在两方面：一是方法的重载上，包括成员方法和构造方法的重载；二是在继承过程中，方法的重写。

多态性是面向对象的重要特征。方法重载和方法覆写实际上属于多态性的一种体现，真正的多态性还包括对象多态性的概念。

对象多态性主要是指子类和父类对象的相互转换关系。

a) 向上类型转换（**upcast**）：比如说将 **Cat** 类型转换为 **Animal** 类型，即将子类型转换为父类型。对于向上类型转换，不需要显式指定。

b) 向下类型转换（**downcast**）：比如将 **Animal** 类型转换为 **Cat** 类型。即将父类型转换为子类型。对于向下类型转换，必须要显式指定（必须要使用强制类型转换）。

网上摘抄的一段多态小总结：

1. **Java** 中除了 **static** 和 **final** 方法外，其他所有的方法都是运行时绑定的。在我另外一篇文章中说到 **private** 方法都被隐式指定为 **final** 的，因此 **final** 的方法不会在运行时绑定。当在派生类中重写基类中 **static**、**final**、或 **private** 方法时，实质上是创建了一个新的方法。

2. 在派生类中，对于基类中的 **private** 方法，最好采用不同的名字。

3. 包含抽象方法的类叫做抽象类。注意定义里面包含这样的意思，只要类中包含一个抽象方法，该类就是抽象类。抽象类在派生中就是作为基类的角色，为不同的子类提供通用的接口。

4. 对象清理的顺序和创建的顺序相反，当然前提是自己想手动清理对象，因

为大家都知道 Java 垃圾回收器。

5.在基类的构造方法中小心调用基类中被重写的方法，这里涉及到对象初始化顺序。

6.构造方法是被隐式声明为 `static` 方法。

7.用继承表达行为间的差异，用字段表达状态上的变化。

七、大谈抽象

抽象类（**abstract class**）：使用了 **abstract** 关键字所修饰的类叫做抽象类。

抽象类无法实例化，也就是说，不能 **new** 出来一个抽象类的对象（实例）。

抽象方法（**abstract method**）：使用 **abstract** 关键字所修饰的方法叫做抽象方法。抽象方法需要定义在抽象类中。相对于抽象方法，之前所定义的方法叫做具体方法（有声明，有实现）。

如果一个类包含了抽象方法，那么这个类一定是抽象类。

如果某个类是抽象类，那么该类可以包含具体方法（有声明、有实现）。

如果一个类中包含了抽象方法，那么这个类一定要声明成 **abstract class**，也就是说，该类一定是抽象类；反之，如果某个类是抽象类，那么该类既可以包含抽象方法，也可以包含具体方法。

无论何种情况，只要一个类是抽象类，那么这个类就无法实例化。

在子类继承父类（父类是个抽象类）的情况下，那么该子类必须要实现父类中所定义的所有抽象方法；否则，该子类需要声明成一个 **abstract class**。

八：最后谈接口

Java 语言不支持一个类有多个直接的父类(多继承),但现实例子中，又有很多类似于多继承的例子，比如教师，他的父类既可以是人，也可以是父母，

所以,在 **java** 中就用继承来填充这个空缺,java 不可以多继承, 但可以实现 (**implements**) 多个接口,间接的实现了多继承。

Java 接口的特征归纳:

1, **Java** 接口中的成员变量默认都是 **public,static,final** 类型的(都可省略),必须被显示初始化,即接口中的成员变量为常量(大写,单词之间用"_"分隔)

2, **Java** 接口中的方法默认都是 **public,abstract** 类型的(都可省略),没有方法体,不能被实例化

3, **Java** 接口中只能包含 **public,static,final** 类型的成员变量和 **public,abstract** 类型的成员方法

4, 接口中没有构造方法,不能被实例化

5, 一个接口不能实现(**implements**)另一个接口,但它可以继承多个其它的接口

6, **Java** 接口必须通过类来实现它的抽象方法

```
public class A implements B{...}
```

7, 当类实现了某个 **Java** 接口时,它必须实现接口中的所有抽象方法,否则这个类必须声明为抽象的

8, 不允许创建接口的实例(实例化),但允许定义接口类型的引用变量,该引用变量引用实现了这个接口的类的实例

9, 一个类只能继承一个直接的父类,但可以实现多个接口,间接的实现了多继承.

10、通过接口,可以方便地对已经存在的系统进行自下而上的抽象,对于任意两个类,不管它们是否属于同一个父类,只有它们存在相同的功能,就能从中抽

象出一个接口类型.对于已经存在的继承树,可以方便的从类中抽象出新的接口,但从类中抽象出新的抽象类却不那么容易,因此接口更有利于软件系统的维护与重构.对于两个系统,通过接口交互比通过抽象类交互能获得更好的松耦合.

11、接口是构建松耦合软件系统的重要法宝,由于接口用于描述系统对外提供的所有服务,因此接口中的成员变量和方法都必须是 `public` 类型的,确保外部使用者能访问它们,接口仅仅描述系统能做什么,但不指明如何去做,所有接口中的方法都是抽象方法,接口不涉及和任何具体实例相关的细节,因此接口没有构造方法,不能被实例化,没有实例变量.

二, 比较抽象类与接口

相同点

- 1, 代表系统的抽象层,当一个系统使用一颗继承树上的类时,应该尽量把引用变量声明为继承树的上层抽象类型,这样可以提高两个系统之间的松耦合
- 2, 都不能被实例化
- 3, 都包含抽象方法,这些抽象方法用于描述系统能提供哪些服务,但不提供具体的实现

不同点:

- 1, 在抽象类中可以为部分方法提供默认的实现,从而避免在子类中重复实现它们,这是抽象类的优势,但这一优势限制了多继承,而接口中只能包含抽象方法.由于在抽象类中允许加入具体方法,因此扩展抽象类的功能,即向抽象类中添加具体方法,不会对它的子类造成影响,而对于接口,一旦接口被公布,就必

须非常稳定,因为随意在接口中添加抽象方法,会影响到所有的实现类,这些实现类要么实现新增的抽象方法,要么声明为抽象类

2, 一个类只能继承一个直接的父类,这个父类可能是抽象类,但一个类可以实现多个接口,这是接口的优势,但这一优势是以不允许为任何方法提供实现作为代价的 三, 为什么 Java 语言不允许多重继承呢?当子类覆盖父类的实例方法或隐藏父类的成员变量及静态方法时,Java 虚拟机采用不同的绑定规则,假如还允许一个类 有多个直接的父类,那么会使绑定规则更加复杂,

结论:

因此,为了简化系统结构设计和动态绑定机制,Java 语言禁止多重继承.而接口中只有抽象方法,没有实例变量和静态方法,只有接口的实现类才会实现 接口的抽象方法(接口中的抽象方法是通过类来实现的),因此,一个类即使有多个接口,也不会增加 Java 虚拟机进行动态绑定的复杂度.因为 Java 虚拟机 永远不会把方法与接口绑定,而只会把方法与它的实现类绑定.四, 使用接口和抽象类的总体原则:

1, 用接口作为系统与外界交互的窗口站在外界使用者(另一个系统)的角度,接口向使用者承诺系统能提供哪些服务,站在系统本身的角度,接口制定系统必须实现哪些服务,接口是系统中最高层次的抽象类型.通过接口交互可以提高两个系统之间的松耦合系统 A 通过系统 B 进行交互,是指系统 A 访问系统 B 时,把引用变量声明 为系统 B 中的接口类型,该引用变量引用系统 B 中接口的实现类的实例。

2, Java 接口本身必须非常稳定,Java 接口一旦制定,就不允许随意更改,否则对外面使用者及系统本身造成影响

3, 用抽象类来定制系统中的扩展点

抽象类来完成部分实现,还要一些功能通过它的子类来实现

(四) java 开发常用类（包装，数字处理集合等）（上）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

一：首谈 java 中的包装类

Java 为基本类型提供包装类，这使得任何接受对象的操作也可以用来操作基本类型，直接将简单类型的变量表示为一个类，在执行变量类型的相互转换时，我们会大量使用这些包装类。java 是一种面向对象语言,java 中的类把方法与数据连接在一起,并构成了自包含式的处理单元.但在 java 中不能定义基本类型(primitive type),为了能将基本类型视为对象来处理,并能连接相关的方法,java 为每个基本类型都提供了包装类,这样,我们便可以把这些基本类型转化为对象来处理了.这些包装类

有: **Boolean,Byte,Short,Character,Integer,Long,Float,Void** 等

值得说明的是,java 是可以直接处理基本类型的,但是在有些情况下我们需要将其作为对象来处理,这时就需要将其转化为包装类了.所有的包装类

(Wrapper Class)都有共同的方法,他们是:

(1)带有基本值参数并创建包装类对象的构造函数.如可以利用 Integer 包装类创建对象,Integer obj=new Integer(145);

(2)带有字符串参数并创建包装类对象的构造函数.如 new Integer("-45.36");

(3)生成字符串表示法的 toString()方法,如 obj.toString().

- (4)对同一个类的两个对象进行比较的 equals()方法,如 obj1.equals(obj2);
- (5)生成哈希表代码的 hashCode 方法,如 obj.hashCode();
- (6)将字符串转换为基本值的 parseInt 方法,如 Integer.parseInt(args[0]);
- (7)可生成对象基本值的 intValue 方法,如 obj.intValue();

在一定的场合,运用 java 包装类来解决问题,能大大提高编程效率.

包装类的自动装箱，自动拆箱

所谓装箱，就是把基本类型用它们相对应的引用类型包起来，使它们可以具有对象的特质，如我们可以把 int 型包装成 Integer 类的对象，或者把 double 包装成 Double，等等。

所谓拆箱，就是跟装箱的方向相反，将 Integer 及 Double 这样的引用类型的对象重新简化为值类型的数据

javaSE5.0 后提供了自动装箱与拆箱的功能，此功能事实上是编译器来帮您的忙，编译器在编译时期依您所编写的方法，决定是否进行装箱或拆箱动作。

自动装箱的过程：每当需要一种类型的对象时，这种基本类型就自动地封装到与它相同类型的包装中。

自动拆箱的过程：每当需要一个值时，被装箱对象中的值就被自动地提取出来，没必要再去调用 intValue()和 doubleValue()方法。

自动装箱，只需将该值赋给一个类型包装器引用，java 会自动创建一个对象。

例如：Integer i=100;//没有通过使用 new 来显示建立，java 自动完成。

自动拆箱，只需将该对象值赋给一个基本类型即可，例如

· int i = 11;

· Integer j = i; //自动装箱

· int k = j //自动拆箱

然而在 Integer 的自动装拆箱会有些细节值得注意:

```
public static void main(String[] args) {  
    Integer a=100;  
    Integer b=100;  
    Integer c=200;  
    Integer d=200;  
    System.out.println(a==b); //1  
    System.out.println(a==100); //2  
    System.out.println(c==d); //3  
    System.out.println(c==200); //4  
}
```

在 java 种, "=="是比较 object 的 reference 而不是 value, 自动装箱后, abcd 都是 Integer 这个 Object, 因此“==”比较的是其引用。按照常规思维, 1 和 3 都应该输出 false。但结果是:

```
true  
true  
false  
true
```

结果 2 和 4, 是因为 ac 进行了自动拆箱, 因此其比较是基本数据类型的比较, 就跟 int 比较时一样的, “==”在这里比较的是它们的值, 而不是引用。

对于结果 1, 虽然比较的时候, 还是比较对象的 reference,但是自动装箱时,

java 在编译的时候 Integer a = 100; 被翻译成

-> Integer a = Integer.valueOf(100);

关键就在于这个 valueOf()的方法。

```
public static Integer valueOf(int i) {  
    final int offset = 128;  
    if (i >= -128 && i <= 127) { // must cache
```

```

return IntegerCache.cache[i + offset];
}
return new Integer(i);
}
private static class IntegerCache {
    private IntegerCache(){}
    static final Integer cache[] = new Integer[-(-128) + 127 + 1];
    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Integer(i - 128);
    }
}

```

根据上面的 jdk 源码，java 为了提高效率，IntegerCache 类中有一个数组缓存了值从-128 到 127 的 Integer 对象。当我们调用 Integer.valueOf (int i) 的时候，如果 i 的值是 >=-128 且 <=127 时，会直接从这个缓存中返回一个对象，否则就 new 一个 Integer 对象。

具体如下：

```

static final Integer cache[] = new Integer[-(-128) + 127 + 1]; //将 cache[] 变成静态
static {
    for(int i = 0; i < cache.length; i++)
        cache[i] = new Integer(i - 128); //初始化 cache[i]
}

```

这是用一个 for 循环对数组 cache 赋值，cache[255] = new Integer(255-128), 也就是 new 一个 Integer(127), 并把引用赋值给 cache[255], 好了，然后是 Integer b= 127, 流程基本一样，最后又到了 cache[255] = new Integer(255-128), 这一句，那我们迷糊了，这不是又 new 了一个对象 127 吗，然后把引用赋值给 cache[255], 我们比较这两个引用（前面声明 a 的时候也有一个）, 由于

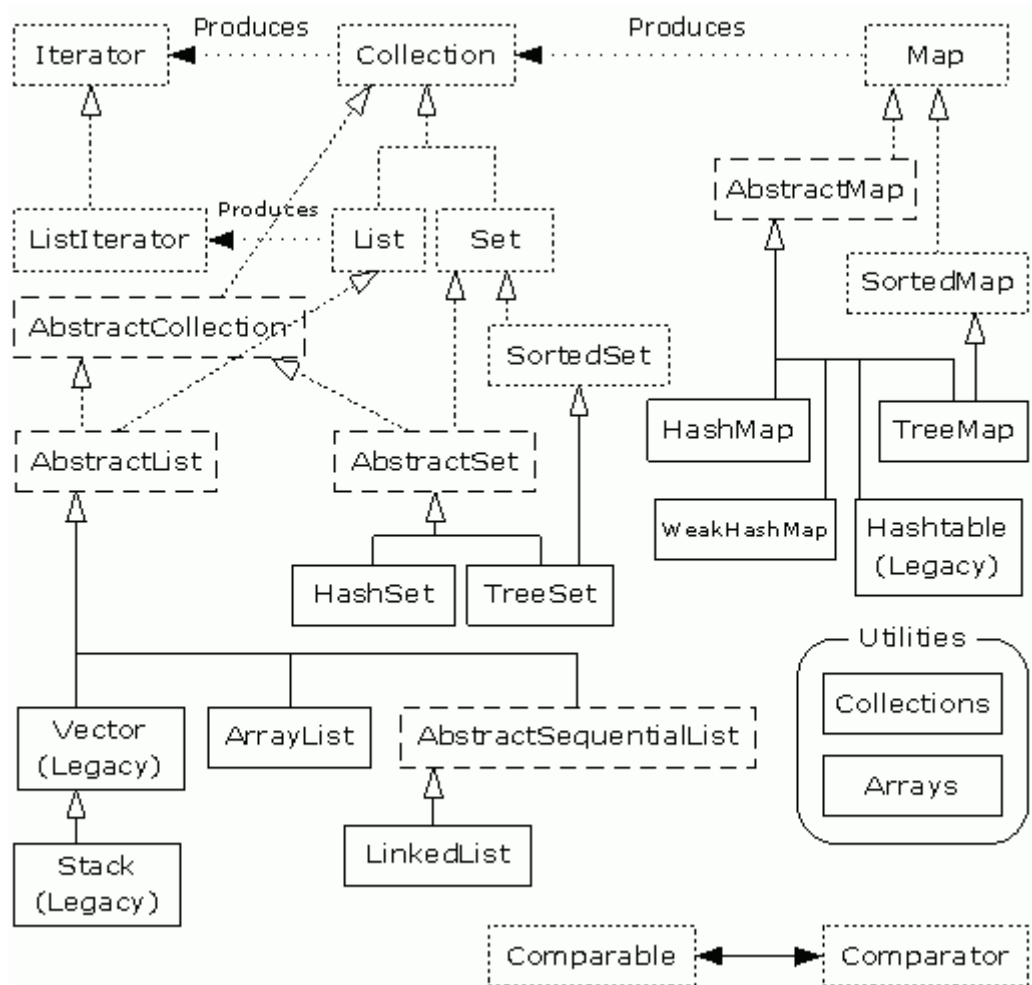
是不同的地址，所以肯定不会相等，应该返回 `false` 啊！呵呵，这么想你就错了，请注意看 `for` 语句给 `cache[i]` 初始化的时候外面还有一个 `{}` 呢，`{}` 前面一个大大的 `static` 关键字，是静态的，那么我们就可以回想下 `static` 有什么特性了，只能 初始化一次，在对象间共享，也就是不同的对象共享同一个 `static` 数据

。那么当我们 `Integer b = 127` 的时候，并没有 `new` 出一个新对象来，而是共享了 `a` 这个对象的引用，记住，他们共享了同一个引用！！！！，那么我们进行比较 `a==b` 时，由于是同一个对象的引用（她们在堆中的地址相同），那当然返回 `true` 了！！！！

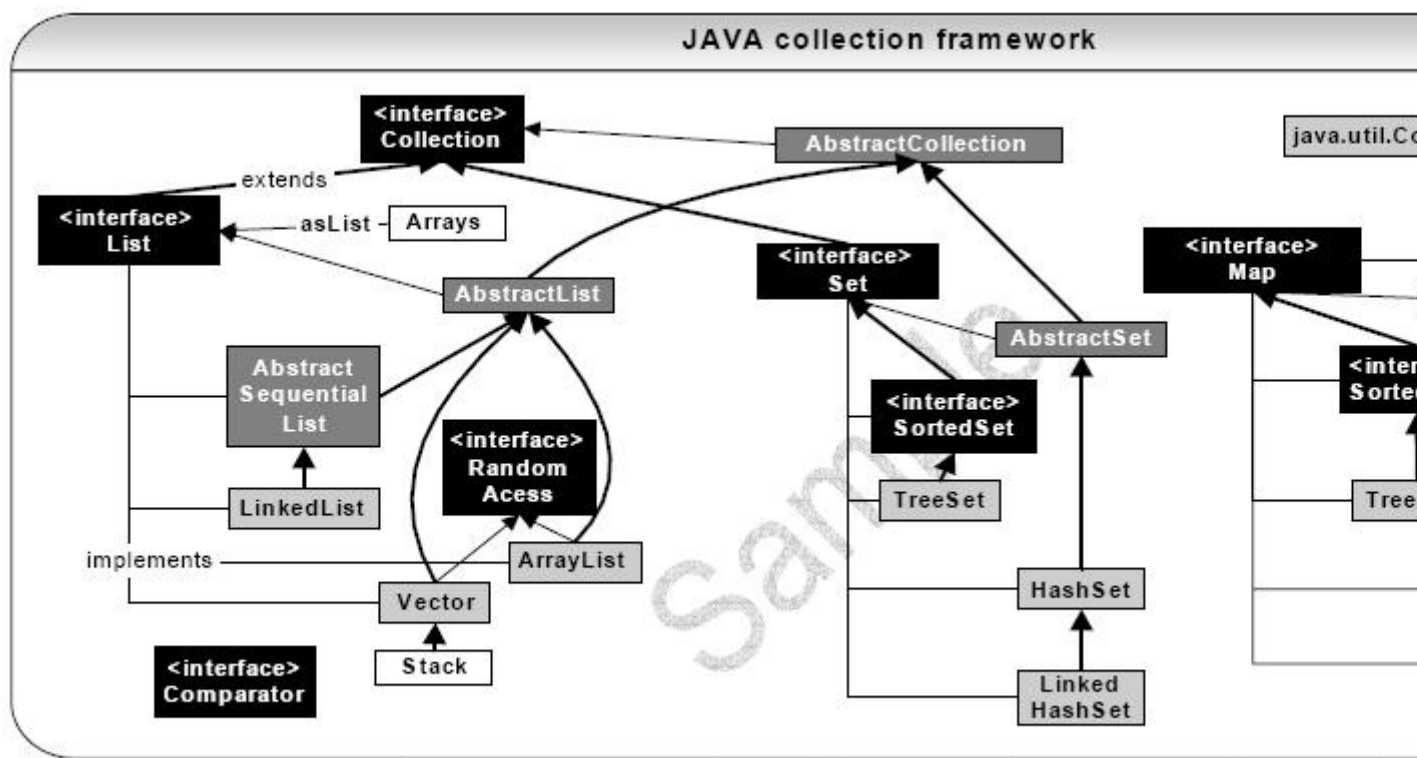
二：进军集合类

集合其实就是存放对象的容器，专业点说就是集合是用来存储和管理其他对象的对象，即对象的容器。集合可以扩容，长度可变，可以存储多种类型的数据，而数组长度不可变，只能存储单一类型的元素

用一张图来总结一下集合的总况：



下面是网上找的一个图片：



(Diagram sourced from: <http://www.wilsonmar.com/1arrays.htm>)

集合中的结构和几个实现类：

		有序否	允许元素重复否
Collection		否	是
List		是	是
Set	AbstractSet	否	否
	HashSet	是（用二叉树排序）	
	TreeSet	是（用二叉树排序）	
Map	AbstractMap	否	使用key-value来映射和存储数据，Key必须惟一，value可以重复
	HashMap	是（用二叉树排序）	
	TreeMap	是（用二叉树排序）	

总述： List、Set、Map 是这个集合体系中最主要的三个接口。

其中 List 和 Set 继承自 Collection 接口。

Set 不允许元素重复。HashSet 和 TreeSet 是两个主要的实现类。

List 有序且允许元素重复。ArrayList、LinkedList 和 Vector 是三个主要的实现类。

Map 也属于集合系统，但和 Collection 接口不同。Map 是 key 对 value 的映

射集合，其中 key 列就是一个集合。key 不能重复，但是 value 可以重复。

HashMap、TreeMap 和 Hashtable 是三个主要的实现类。

SortedSet 和 SortedMap 接口对元素按指定规则排序，SortedMap 是对 key 列进行排序。

具体来说：

1.Collection 接口 用于表示任何对象或元素组。想要尽可能以常规方式处理一组元素时，就使用这一接口。

操作：

(1) 单元素添加、删除操作：

`boolean add(Object o)`: 将对象添加给集合

`boolean remove(Object o)`: 如果集合中有与 o 相匹配的对象，则删除对象 o

(2) 查询操作：

`int size()` : 返回当前集合中元素的数量

`boolean isEmpty()` : 判断集合中是否有任何元素

`boolean contains(Object o)` : 查找集合中是否含有对象 o

`Iterator iterator()` : 返回一个迭代器，用来访问集合中的各个元素

(3) 组操作 : 作用于元素组或整个集合

`boolean containsAll(Collection c)`: 查找集合中是否含有集合 `c` 中所有元素

`boolean addAll(Collection c)`: 将集合 `c` 中所有元素添加给该集合

`void clear()`: 删除集合中所有元素

`void removeAll(Collection c)`: 从集合中删除集合 `c` 中的所有元素

`void retainAll(Collection c)`: 从集合中删除集合 `c` 中不包含的元素

(4) `Collection` 转换为 `Object` 数组 :

`Object[] toArray()`: 返回一个内含集合所有元素的 `array`

`Object[] toArray(Object[] a)`: 返回一个内含集合所有元素的 `array`。运行期返回的 `array` 和参数 `a` 的型别相同，需要转换为正确型别。

此外，您还可以把集合转换成其它任何其它的对象数组。但是，您不能直接把集合转换成基本数据类型的数组，因为集合必须持有对象。

“斜体接口方法是可选的。因为一个接口实现必须实现所有接口方法，调用程序就需要一种途径来知道一个可选的方法是不是不受支持。如果调用一种可选方法 时，一个 `UnsupportedOperationException` 被抛出，则操作失败，因为方法不受支持。此异常类继承 `RuntimeException` 类，避免了将所有集合操作放入 `try-catch` 块。”

Collection 不提供 get() 方法。如果要遍历 Collection 中的元素,就必须用 Iterator。

2.List 接口对 Collection 进行了简单的扩充,它的具体实现类常用的有

ArrayList 和 LinkedList。你可以将任何东西放到一个 List 容器中,并在需要时从中取出。ArrayList 从其命名中可以看出它是一种类似数组的形式进行存储,因此它的随机访问速度极快,而 LinkedList 的内部实现是链表,它适合于在链表中间需要频繁进行插入和删除操作。在具体应用时可以根据需要自由选择。前面说的 Iterator 只能对容器进行向前遍历,而 ListIterator 则继承了 Iterator 的思想,并提供了对 List 进行双向遍历的方法。

(1) 面向位置的操作包括插入某个元素或 Collection 的功能,还包括获取、除去或更改元素的功能。在 List 中搜索元素可以从列表的头部或尾部开始,如果找到元素,还将报告元素所在的位置 :

void add(int index, Object element): 在指定位置 index 上添加元素 element

boolean addAll(int index, Collection c): 将集合 c 的所有元素添加到指定位置 index

Object get(int index): 返回 List 中指定位置的元素

int indexOf(Object o): 返回第一个出现元素 o 的位置, 否则返回-1

int lastIndexOf(Object o) : 返回最后一个出现元素 o 的位置, 否则返回-1

Object remove(int index) : 删除指定位置上的元素

`Object set(int index, Object element)` : 用元素 `element` 取代位置 `index` 上的元素，并且返回旧的元素

(2) `List` 接口不但以位置序列迭代的遍历整个列表，还能处理集合的子集：

`ListIterator listIterator()` : 返回一个列表迭代器，用来访问列表中的元素

`ListIterator listIterator(int index)` : 返回一个列表迭代器，用来从指定位置 `index` 开始访问列表中的元素

`List subList(int fromIndex, int toIndex)` : 返回从指定位置 `fromIndex`（包含）到 `toIndex`（不包含）范围中各个元素的列表视图

“对子列表的更改（如 `add()`、`remove()` 和 `set()` 调用）对底层 `List` 也有影响。”“`ArrayList` 和 `LinkedList` 都实现 `Cloneable` 接口，都提供了两个构造函数，一个无参的，一个接受另一个 `Collection`”

LinkedList 类

`LinkedList` 类添加了一些处理列表两端元素的方法。

(1) `void addFirst(Object o)`: 将对象 `o` 添加到列表的开头

`void addLast(Object o)`: 将对象 `o` 添加到列表的结尾

(2) `Object getFirst()`: 返回列表开头的元素

`Object getLast()`: 返回列表结尾的元素

(3) `Object removeFirst()`: 删除并且返回列表开头的元素

`Object removeLast()`: 删除并且返回列表结尾的元素

(4) `LinkedList()`: 构建一个空的链接列表

`LinkedList(Collection c)`: 构建一个链接列表，并且添加集合 `c` 的所有元素

“使用这些新方法，您就可以轻松的把 `LinkedList` 当作一个堆栈、队列或其它面向端点的数据结构。”

ArrayList 类

`ArrayList` 类封装了一个动态再分配的 `Object[]` 数组。每个 `ArrayList` 对象有一个 `capacity`。这个 `capacity` 表示存储列表中元素的数组的容量。当元素添加到 `ArrayList` 时，它的 `capacity` 在常量时间内自动增加。

在向一个 `ArrayList` 对象添加大量元素的程序中，可使用 `ensureCapacity` 方法增加 `capacity`。这可以减少增加重分配的数量。

(1) `void ensureCapacity(int minCapacity)`: 将 `ArrayList` 对象容量增加 `minCapacity`

(2) `void trimToSize()`: 整理 `ArrayList` 对象容量为列表当前大小。程序可使用这个操作减少 `ArrayList` 对象存储空间。

（五）java 开发常用类（包装，数字处理集合等）（下）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

写在前面：由于前天项目老师建设局的项目快到验收阶段，所以，前天晚上通宵，昨天睡了大半天，下午我们宿舍聚会，所以时间有点耽误，希望大家见谅

3.Set 接口也是 Collection 的一种扩展，而与 List 不同的时，在 Set 中的对象元素不能重复，也就是说你不能把同样的东西两次放入同一个 Set 容器中。它的常用具体实现有 HashSet 和 TreeSet 类。HashSet 能快速定位一个元素，但是你放到 HashSet 中的对象需要实现 hashCode()方法，它使用了前面说过的哈希码的算法。而 TreeSet 则将放入其中的元素按序存放，这就要求你放入其中的对象是可排序的，这就用到了集合框架提供的另外两个实用类 Comparable 和 Comparator。一个类是可排序的，它就应该实现 Comparable 接口。有时多个类具有相同的排序算法，那就不需要在每分别重复定义相同的排序算法，只要实现 Comparator 接口即可。集合框架中还有两个很实用的公用类：Collections 和 Arrays。Collections 提供了对一个 Collection 容器进行诸如排序、复制、查找和填充等一些非常实用的方法，Arrays 则是对一个数组进行类似的操作。

Hash 表

Hash 表是一种数据结构，用来查找对象。Hash 表为每个对象计算出一个整数，称为 Hash Code(哈希码)。Hash 表是个链接式列表的阵列。每个列表称为一个 buckets(哈希表元)。对象位置的计算

$$\text{index} = \text{HashCode} \% \text{buckets}$$
(HashCode 为对象哈希码，buckets 为哈希表元总数)。

当你添加元素时，有时你会遇到已经填充了元素的哈希表元，这种情况称为 Hash Collisions(哈希冲突)。这时，你必须判断该元素是否已经存在于该哈希表中。

如果哈希码是合理地随机分布的，并且哈希表元的数量足够大，那么哈希冲突的数量就会减少。同时，你也可以通过设定一个初始的哈希表元数量来更好地控制哈希表的运行。初始哈希表元的数量为

$\text{buckets} = \text{size} * 150\% + 1$ (size 为预期元素的数量)。

如果哈希表中的元素放得太满，就必须进行 rehashing(再哈希)。再哈希使哈希表元数增倍，并将原有的对象重新导入新的哈希表元中，而原始的哈希表元被删除。load factor(加载因子)决定何时要对哈希表进行再哈希。在 Java 编程语言中，加载因子默认值为 0.75，默认哈希表元为 101。

Comparable 接口和 Comparator 接口

在“集合框架”中有两种比较接口：Comparable 接口和 Comparator 接口。像 String 和 Integer 等 Java 内建类实现 Comparable 接口以提供一定排序方式，但这样只能实现该接口一次。对于那些没有实现 Comparable 接口的类、或者自定义的类，您可以通过 Comparator 接口来定义您自己的比较方式。

Comparable 接口

在 java.lang 包中，Comparable 接口适用于一个类有自然顺序的时候。假定对象集合是同一类型，该接口允许您把集合排序成自然顺序。

(1) `int compareTo(Object o)`: 比较当前实例对象与对象 o，如果位于对象 o 之前，返回负值，如果两个对象在排序中位置相同，则返回 0，如果位于对象 o 后面，则返回正值

在 Java 2 SDK 版本 1.4 中有二十四个类实现 Comparable 接口。下表展示了 8

种基本类型的自然排序。虽然一些类共享同一种自然排序，但只有相互可比的类才能排序。

类	排序
BigDecimal,BigInteger,Byte, Double, Float,Integer,Long, Short	按数字大小 排序
Character	按 Unicode 值的数字大 小排序
String	按字符串中 字 符 Unicode 值排序

利用 Comparable 接口创建您自己的类的排序顺序，只是实现 compareTo()方法的问题。通常就是依赖几个数据成员的自然排序。同时类也应该覆盖 equals()和 hashCode()以确保两个相等的对象返回同一个哈希码。

Comparator 接口

若一个类不能用于实现 java.lang.Comparable，或者您不喜欢缺省的 Comparable 行为并想提供自己的排序顺序(可能多种排序方式)，你可以实现 Comparator 接口，从而定义一个比较器。

(1)int compare(Object o1, Object o2): 对两个对象 o1 和 o2 进行比较，如果 o1 位于 o2 的前面，则返回负值，如果在排序顺序中认为 o1 和 o2 是相同的，

返回 0，如果 o1 位于 o2 的后面，则返回正值

“与 Comparable 相似，0 返回值不表示元素相等。一个 0 返回值只是表示两个对象排在同一位置。由 Comparator 用户决定如何处理。如果两个不相等的元素比较的结果为零，您首先应该确信那就是您要的结果，然后记录行为。”

(2)boolean equals(Object obj): 指示对象 obj 是否和比较器相等。

“该方法覆写 Object 的 equals()方法，检查的是 Comparator 实现的等同性，不是处于比较状态下的对象。”

SortedSet 接口

“集合框架”提供了个特殊的 Set 接口：SortedSet，它保持元素的有序顺序。

SortedSet 接口为集的视图(子集)和它的两端（即头和尾） 提供了访问方法。当您处理列表的子集时，更改视图会反映到源集。此外，更改源集也会反映在子集上。发生这种情况的原因在于视图由两端的元素而不是下标元素 指定，所以如果您想要一个特殊的高端元素（toElement）在子集中，您必须找到下一个元素。

添加到 SortedSet 实现类的元素必须实现 Comparable 接口，否则您必须给它的构造函数提供一个 Comparator 接口的实现。TreeSet 类是它的唯一一份实现。

“因为集必须包含唯一的项，如果添加元素时比较两个元素导致了 0 返回值（通过 Comparable 的 compareTo()方法或 Comparator 的 compare()方法），那么新元素就没有添加进去。如果两个元素相等，那还好。但如果它们不相等

的话，您接下来就应该修改比较方法，让比较方法和 `equals()` 的效果一致。”

(1) `Comparator comparator()`: 返回对元素进行排序时使用的比较器，如果使用 `Comparable` 接口的 `compareTo()` 方法对元素进行比较，则返回 `null`

(2) `Object first()`: 返回有序集合中第一个(最低)元素

(3) `Object last()`: 返回有序集合中最后一个(最高)元素

(4) `SortedSet subSet(Object fromElement, Object toElement)`: 返回从 `fromElement`(包括)至 `toElement`(不包括)范围内元素的 `SortedSet` 视图(子集)

(5) `SortedSet headSet(Object toElement)`: 返回 `SortedSet` 的一个视图，其内各元素皆小于 `toElement`

(6) `SortedSet tailSet(Object fromElement)`: 返回 `SortedSet` 的一个视图，其内各元素皆大于或等于 `fromElement`

AbstractSet 抽象类

`AbstractSet` 类覆盖了 `Object` 类的 `equals()` 和 `hashCode()` 方法，以确保两个相等的集返回相同的哈希码。若两个集大小相等 且包含相同元素，则这两个集相等。按定义，集的哈希码是集中元素哈希码的总和。因此，不论集的内部顺序如何，两个相等的集会有相同的哈希码。

HashSet 类和 TreeSet 类

“集合框架”支持 `Set` 接口两种普通的实现：`HashSet` 和 `TreeSet`(`TreeSet` 实现 `SortedSet` 接口)。在更多情况下，您会使用 `HashSet` 存储重复自由的集合。考虑到效率，添加到 `HashSet` 的对象需要采用恰当分配哈希码的方式来实

现 hashCode()方法。虽然大多数系统类覆盖了 Object 中缺省的 hashCode() 和 equals()实现，但创建您自己的要添加到 HashSet 的类时，别忘了覆盖 hashCode()和 equals()。

当您要从集合中以有序的方式插入和抽取元素时，TreeSet 实现会有用处。为了能顺利进行，添加到 TreeSet 的元素必须是可排序的。

HashSet 类

(1) HashSet(): 构建一个空的哈希集

(2) HashSet(Collection c): 构建一个哈希集，并且添加集合 c 中所有元素

(3) HashSet(int initialCapacity): 构建一个拥有特定容量的空哈希集

(4) HashSet(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空哈希集。LoadFactor 是 0.0 至 1.0 之间的一个数

TreeSet 类

(1) TreeSet():构建一个空的树集

(2) TreeSet(Collection c): 构建一个树集，并且添加集合 c 中所有元素

(3) TreeSet(Comparator c): 构建一个树集，并且使用特定的比较器对其元素进行排序

“comparator 比较器没有任何数据，它只是比较方法的存放器。这种对象有时称为函数对象。函数对象通常在“运行过程中”被定义为匿名内部类的一个实例。”

TreeSet(SortedSet s): 构建一个树集，添加有序集合 s 中所有元素，并且使用与有序集合 s 相同的比较器排序

LinkedHashSet 类

LinkedHashSet 扩展 **HashSet**。如果想跟踪添加给 **HashSet** 的元素的顺序，**LinkedHashSet** 实现会有帮助。**LinkedHashSet** 的迭代器按照元素的插入顺序来访问各个元素。它提供了一个可以快速访问各个元素的有序集合。同时，它也增加了实现的代价，因为 哈希表元中的各个元素是通过双重链接式列表链接在一起的。

(1) **LinkedHashSet():** 构建一个空的链接式哈希集

(2) **LinkedHashSet(Collection c):** 构建一个链接式哈希集，并且添加集合 c 中所有元素

(3) **LinkedHashSet(int initialCapacity):** 构建一个拥有特定容量的空链接式哈希集

(4) **LinkedHashSet(int initialCapacity, float loadFactor):** 构建一个拥有特定容量和加载因子的空链接式哈希集。LoadFactor 是 0.0 至 1.0 之间的一个数

“为优化 HashSet 空间的使用，您可以调优初始容量和负载因子。TreeSet 不含调优选项，因为树总是平衡的。”

4.Map 是一种把键对象和值对象进行关联的容器，而一个值对象又可以是一个 Map，依次类推，这样就可形成一个多级映射。对于键对象来说，像 Set 一样，一个 Map 容器中的键对象不允许重复，这是为了保持查找结果的一致性；如果有两个键对象一样，那你想得到那个键对象所对应的值对象时就有问题了，可能你得到的并不是你想的那个值对象，结果会造成混乱，所以键的唯一性很重要，也是符合集合的性质的。当然在使用过程中，某个键所对应的值对象可能会发生变化，这时会按照最后一次修改的值对象与键对应。对于值对象则没有唯一性的要求。你可以将任意多个键都映射到一个值对象上，这不会发生任何问题（不过对你的使用却可能会造成不便，你不知道你得到的到底是那一个键所对应的值对象）。Map 有两种比较常用的实现：HashMap 和 TreeMap。HashMap 也用到了哈希码的算法，以便快速查找一个键，TreeMap 则是对键按序存放，因此它便有一些扩展的方法，比如 firstKey(),lastKey()等，你还可以从 TreeMap 中指定一个范围以取得其子 Map。键和值的关联很简单，用 put(Object key,Object value)方法即可将一个键与一个值对象相关联。用 get(Object key)可得到与此 key 对象所对应的值对象。

Map 接口不是 Collection 接口的继承。Map 接口用于维护键/值对 (key/value pairs)。该接口描述了从不重复的键到值的映射。

(1) 添加、删除操作：

Object put(Object key, Object value): 将互相关联的一个关键字与一个值放入该映像。如果该关键字已经存在，那么与此关键字相关的新值将取代旧值。方法返回关键字的旧值，如果关键字原先并不存在，则返回 **null**

Object remove(Object key): 从映像中删除与 **key** 相关的映射

void putAll(Map t): 将来自特定映像的所有元素添加给该映像

void clear(): 从映像中删除所有映射

“键和值都可以为 **null**。但是，您不能把 **Map** 作为一个键或值添加给自身。”

(2) 查询操作：

Object get(Object key): 获得与关键字 **key** 相关的值，并且返回与关键字 **key** 相关的对象，如果没有在该映像中找到该关键字，则返回 **null**

boolean containsKey(Object key): 判断映像中是否存在关键字 **key**

boolean containsValue(Object value): 判断映像中是否存在值 **value**

int size(): 返回当前映像中映射的数量

boolean isEmpty(): 判断映像中是否有任何映射

(3) 视图操作 ： 处理映像中键/值对组

Set keySet(): 返回映像中所有关键字的视图集

“因为映射中键的集合必须是唯一的，您用 **Set** 支持。您还可以从视图中删除元素，同时，关键字和它相关的值将从源映像中被删除，但是你不能添加任何元素。”

Collection values():返回映像中所有值的视图集

“因为映射中值的集合不是唯一的，您用 **Collection** 支持。您还可以从视图中删除元素，同时，值和它的关键字将从源映像中被删除，但是你不能添加任何元素。”

Set entrySet(): 返回 **Map.Entry** 对象的视图集，即映像中的关键字/值对

“因为映射是唯一的，您用 **Set** 支持。您还可以从视图中删除元素，同时，这些元素将从源映像中被删除，但是你不能添加任何元素。”

Map.Entry 接口

Map 的 **entrySet()**方法返回一个实现 **Map.Entry** 接口的对象集合。集合中每个对象都是底层 **Map** 中一个特定的键/值对。

通过这个集合的迭代器，您可以获得每一个条目(唯一获取方式)的键或值并对值进行更改。当条目通过迭代器返回后，除非是迭代器自身的 **remove()** 方法或者迭代器返回的条目的 **setValue()**方法，其余对源 **Map** 外部的修改都会导致此条目集变得无效，同时产生条目行为未定义。

(1) `Object getKey()`: 返回条目的关键字

(2) `Object getValue()`: 返回条目的值

(3) `Object setValue(Object value)`: 将相关映像中的值改为 `value`，并且返回旧值

SortedMap 接口

“集合框架”提供了个特殊的 `Map` 接口：`SortedMap`，它用来保持键的有序顺序。

`SortedMap` 接口为映像的视图(子集)，包括两个端点提供了访问方法。除了排序是作用于映射的键以外，处理 `SortedMap` 和处理 `SortedSet` 一样。

添加到 `SortedMap` 实现类的元素必须实现 `Comparable` 接口，否则您必须给它的构造函数提供一个 `Comparator` 接口的实现。`TreeMap` 类是它的唯一一份实现。

“因为对于映射来说，每个键只能对应一个值，如果在添加一个键/值对时比较两个键产生了 0 返回值（通过 `Comparable` 的 `compareTo()` 方法或通过 `Comparator` 的 `compare()` 方法），那么，原始键对应值被新的值替代。如果两个元素相等，那还好。但如果不相等，那么您就应该修改 比较方法，让比较方法和 `equals()` 的效果一致。”

(1) `Comparator comparator()`: 返回对关键字进行排序时使用的比较器，如果使用 `Comparable` 接口的 `compareTo()` 方法对关键字进行比较，则返回 `null`

(2) `Object firstKey()`: 返回映像中第一个(最低)关键字

(3) `Object lastKey()`: 返回映像中最后一个(最高)关键字

(4) `SortedMap subMap(Object fromKey, Object toKey)`: 返回从 `fromKey`(包括)至 `toKey`(不包括)范围内元素的 `SortedMap` 视图(子集)

(5) `SortedMap headMap(Object toKey)`: 返回 `SortedMap` 的一个视图，其内各元素的 `key` 皆小于 `toKey`

(6) `SortedSet tailMap(Object fromKey)`: 返回 `SortedMap` 的一个视图，其内各元素的 `key` 皆大于或等于 `fromKey`

AbstractMap 抽象类

和其它抽象集合实现相似，`AbstractMap` 类覆盖了 `equals()` 和 `hashCode()` 方法以确保两个相等映射返回相同的哈希码。如果两个映射大小相等、包含同样的键且每个键在这两个映射中对应的值都相同，则这两个映射相等。映射的哈希码是映射元素哈希码的总和，其中每个元素是 `Map.Entry` 接口的一个实现。因此，不论映射内部顺序如何，两个相等映射会报告相同的哈希码。

HashMap 类和 TreeMap 类

“集合框架”提供两种常规的 Map 实现：HashMap 和 TreeMap (TreeMap 实现 SortedMap 接口)。在 Map 中插入、删除和定位元素，HashMap 是最好的选择。但如果您要按自然顺序或自定义顺序遍历键，那么 TreeMap 会更好。使用 HashMap 要求添加的键类明确定义了 hashCode()和 equals()的实现。

这个 TreeMap 没有调优选项，因为该树总处于平衡状态。

HashMap 类

为了优化 HashMap 空间的使用，您可以调优初始容量和负载因子。

(1) HashMap(): 构建一个空的哈希映像

(2) HashMap(Map m): 构建一个哈希映像，并且添加映像 m 的所有映射

(3) HashMap(int initialCapacity): 构建一个拥有特定容量的空的哈希映像

(4) HashMap(int initialCapacity, float loadFactor): 构建一个拥有特定容量和加载因子的空的哈希映像

TreeMap 类

TreeMap 没有调优选项，因为该树总处于平衡状态。

(1) `TreeMap()`: 构建一个空的映像树

(2) `TreeMap(Map m)`: 构建一个映像树，并且添加映像 `m` 中所有元素

(3) `TreeMap(Comparator c)`: 构建一个映像树，并且使用特定的比较器对关键字进行排序

(4) `TreeMap(SortedMap s)`: 构建一个映像树，添加映像树 `s` 中所有映射，并且使用与有序映像 `s` 相同的比较器排序

LinkedHashMap 类

`LinkedHashMap` 扩展 `HashMap`，以插入顺序将关键字/值对添加进链接哈希映像中。象 `LinkedHashSet` 一样，`LinkedHashMap` 内部也采用双重链接式列表。

(1) `LinkedHashMap()`: 构建一个空链接哈希映像

(2) `LinkedHashMap(Map m)`: 构建一个链接哈希映像,并且添加映像 `m` 中所有映射

(3) `LinkedHashMap(int initialCapacity)`: 构建一个拥有特定容量的空的链接哈希映像

(4) `LinkedHashMap(int initialCapacity, float loadFactor)`: 构建一个拥有特定

容量和加载因子的空的链接哈希映像

(5) `LinkedHashMap(int initialCapacity, float loadFactor,`

`boolean accessOrder)`: 构建一个拥有特定容量、加载因子和访问顺序排序的空的链接哈希映像

“如果将 `accessOrder` 设置为 `true`,那么链接哈希映像将使用访问顺序而不是插入顺序来迭

代各个映像。每次调用 `get` 或者 `put` 方法时,相关的映射便从它的当前位置上删除,然后放到链接式映像列表的结尾处(只有链接式映像列表中的位置才会受到影响,哈希表元则不受影响。哈希表映射总是待在对应于关键字的哈希码的哈希表元中)。”

“该特性对于实现高速缓存的“删除最近最少使用”的原则很有用。例如,你可以希望将最常访问的映射保存在内存中,并且从数据库中读取不经常访问的对象。 当你在表中找不到某个映射,并且该表中的映射已经放得非常满时,你可以让迭代器进入该表,将它枚举的开头几个映射删除掉。这些是最近最少使用的映射。”

(6) `protected boolean removeEldestEntry(Map.Entry eldest)`: 如果你想删除最老的映射,则覆盖该方法,以便返回 `true`。当某个映射已经添加给映像之后,便调用该方法。它的默认实现方法返回 `false`,表示默认条件下老的

映射没有被删除。但是你可以重新定义本方法，以便有选择地在最老的映射符合某个条件，或者映像超过了某个大小时，返回 `true`。

（六）I/O 流操作

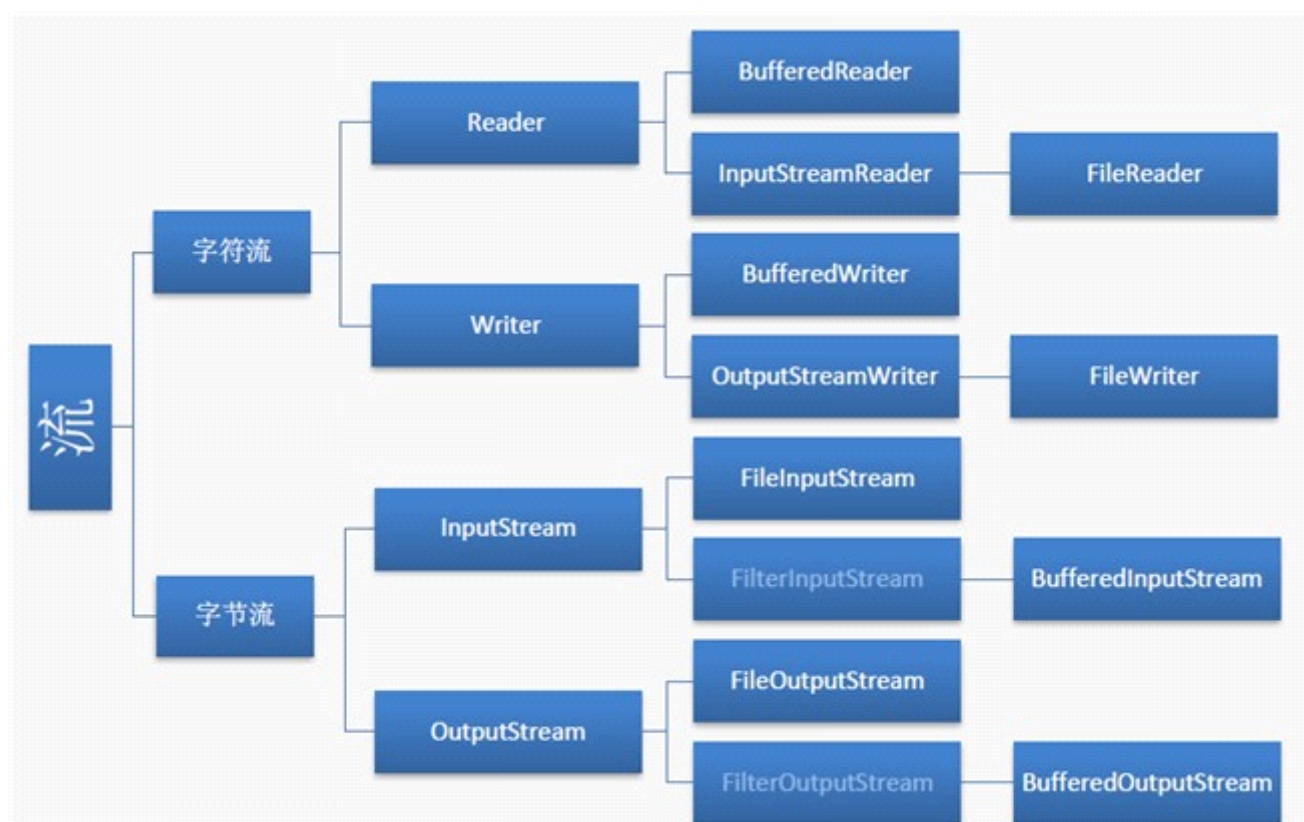
本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

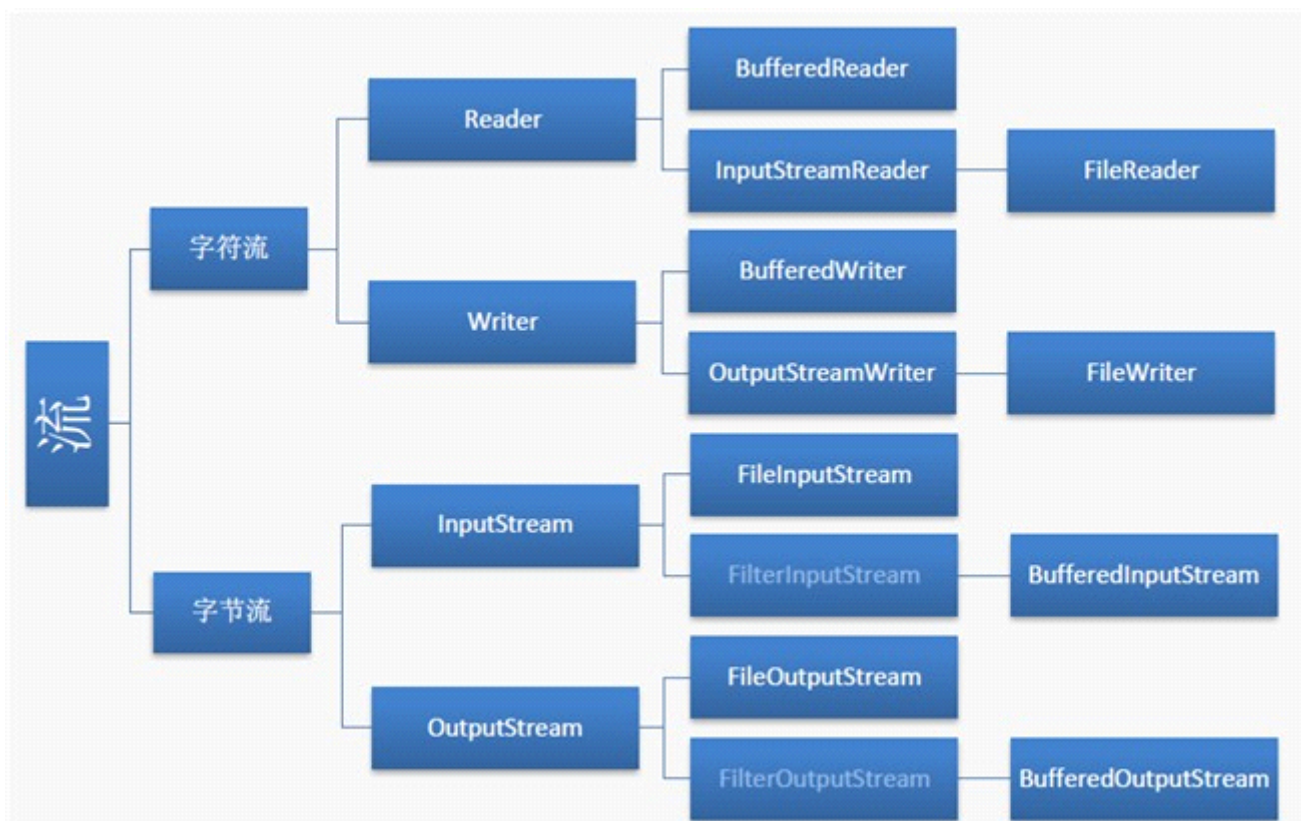
在软件开发中，数据流和数据库操作占据了一个很重要的位置，所以，熟悉操作数据流和数据库，对于每一个开发者来说都是很重要的，今天就来总结一下 I/O，数据库操作

一：从数据流开始

首先先有一个结构图看一下整个数据流中的 API 结构和对象继承关系信息：



其他常用与流有关的对象：



首先从字符流开始

1、字符流的由来：

因为文件编码的不同，而有了对字符进行高效操作的字符流对象。

原理：其实就是基于字节流读取字节时，去查了指定的码表。

字节流和字符流的区别：

1，字节流读取的时候，读到一个字节就返回一个字节。字符流使用了字节流读到一个或多个字节(中文对应的字节数是两个，UTF-8 码表中是 3 个字节)时。先去查指定的编码表，将查到的字符返回。

2，字节流可以处理所有类型数据，如图片，mp3，avi。而字符流只能处理字符数据。

结论：只要是处理纯文本数据，就要优先考虑使用字符流。除此之外都用字节流。

基本的读写操作方式:

因为数据通常都以文件形式存在。

所以就要找到 IO 体系中可以用于操作文件的流对象。

通过名称可以更容易获取该对象。

因为 IO 体系中的子类名后缀绝大部分是父类名称。而前缀都是体现子类功能的名字。

Reader

|--InputStreamReader

|--FileReader:专门用于处理文件的字符读取流对象。

Writer

|--OutputStreamWriter

|--FileWriter:专门用于处理文件的字符写入流对象。

Reader 中的常见的方法:

1, `int read()`: 读取一个字符。返回的是读到的那个字符。如果读到流的末尾, 返回-1.

2, `int read(char[])`: 将读到的字符存入指定的数组中, 返回的是读到的字符个数, 也就是往数组里装的元素的个数。如果读到流的末尾, 返回-1.

3, `close()`:读取字符其实用的是 `window` 系统的功能, 就希望使用完毕后, 进行资源的释放

Writer 中的常见的方法:

1, `write(ch)`: 将一个字符写入到流中。

2, `write(char[])`: 将一个字符数组写入到流中。

3, `write(String)`: 将一个字符串写入到流中。

4, `flush()`:刷新流, 将流中的数据刷新到目的地中, 流还存在。

5, `close()`:关闭资源: 在关闭前会先调用 `flush()`, 刷新流中的数据去目的地。
然流关闭。

FileWriter:该类没有特有的方法只有自己的构造函数。该类特点在于

- 1, 用于处理文本文件。
- 2, 该类中有默认的编码表,
- 3, 该类中有临时缓冲。

构造函数: 在写入流对象初始化时, 必须要有一个存储数据的目的地。

对于读取或者写入流对象的构造函数, 以及读写方法, 还有刷新关闭功能都会抛出 `IOException` 或其子类。所以都要进行处理。或者 `throws` 抛出, 或者 `try catch` 处理

另一个小细节:

当指定绝对路径时, 定义目录分隔符有两种方式:

- 1, 反斜线但是一定要写两个。 `\\new FileWriter("c:\\demo.txt");`
- 2, 斜线/ 写一个即可。 `new FileWriter("c:/demo.txt");`

一个读取文本文件的经典例子:

[java] [view plaincopyprint?](#)

```
1. <SPAN style="COLOR: #000000">FileReader fr = null;
2. try
3. {
4. fr = new FileReader("demo.txt");
5. char[] buf = new char[1024];//该长度通常都是 1024 的整数倍。
6. int len = 0;
7. while((len=fr.read(buf))!=-1)
8. {
9. System.out.println(new String(buf,0,len));
10.}
```



```
11.}  
12.catch(IOException e)  
13.{  
14.System.out.println(e.toString());  
15.}  
16.</SPAN>
```

字符流的缓冲区：缓冲区的出现提高了对流的操作效率。

原理：其实就是将数组进行封装。

对应的对象：

BufferedWriter：特有方法：**newLine()**：跨平台的换行符。

BufferedReader：特有方法：**readLine()**：一次读一行，到行标记时，将行标记之前的字符数据作为字符串返回。当读到末尾时，返回 **null**。

在使用缓冲区对象时，要明确，缓冲的存在是为了增强流的功能而存在，所以在建立缓冲区对象时，要先有流对象存在。

其实缓冲内部就是在使用流对象的方法，只不过加入了数组对数据进行了临时存储。为了提高操作数据的效率。

代码上的体现：

写入缓冲区对象。

//建立缓冲区对象必须把流对象作为参数传递给缓冲区的构造函数。

```
BufferedWriter bufw = new BufferedWriter(new FileWriter("buf.txt"));  
bufw.write("abce");//将数据写入到了缓冲区。
```

```
bufw.flush();//对缓冲区的数据进行刷新。将数据刷到目的地中。
```

```
bufw.close();//关闭缓冲区，其实关闭的是被包装在内部的流对象。
```

读取缓冲区对象。

```
BufferedReader bufr = new BufferedReader(new FileReader("buf.txt"));
String line = null;
//按照行的形式取出数据。取出的每一个行数据不包含回车符。

while((line=bufr.readLine())!=null)
{
    System.out.println(line);
}
bufr.close();
```

readLine():方法的原理:

其实缓冲区中的该方法，用的还是与缓冲区关联的流对象的 **read** 方法。只不过，每一次读到一个字符，先不进行具体操作，先进行临时存储。当读取到回车标记时，将临时容器中存储的数据一次性返回。

既然明确了原理，我们也可以实现一个类似功能的方法。

[java] [view plaincopyprint?](#)

```
1. class MyBufferedReader
2. {
3.     private Reader r;
4.     MyBufferedReader(Reader r)
5.     {
6.         this.r = r;
7.     }
8.     public String myReadLine()throws IOException
9.     { //1,创建临时容器。
10.        StringBuilder sb = new StringBuilder();
11.        //2,循环的使用 read 方法不断读取字符。
12.        int ch = 0;
13.        while((ch=r.read())!=-1)
14.        {
15.            if(ch=='\r')
```

```

16.continue;
17.if(ch=="\n")
18.return sb.toString();
19.else
20.sb.append((char)ch);
21.}
22.if(sb.length()!=0)
23.return sb.toString();
24.return null;
25.}
26.public void myClose()throws IOException
27.{
28.r.close();
29.}
30.}

```

然后说一下字节流：

抽象基类：InputStream，OutputStream。

字节流可以操作任何数据。

注意：字符流使用的数组是字符数组。char [] chs 字节流使用的数组是字节数组。byte [] bt

```
FileOutputStream fos = new FileOutputStream("a.txt");
```

```
fos.write("abcde");//直接将数据写入到了目的地。
```

```
fos.close();//只关闭资源。
```

```
FileInputStream fis = new FileInputStream("a.txt");
```

```
//fis.available();//获取关联的文件的字节数。
```

```
//如果文件体积不是很大。
```

```
//可以这样操作。
```

```
byte[] buf = new byte[fis.available()]; // 创建一个刚刚好的缓冲区。
```

//但是这有一个弊端，就是文件过大，大小超出 jvm 的内容空间时，会内存溢出。

```
fis.read(buf);
```

一个小问题：

字节流的 `read()` 方法读取一个字节。为什么返回的不是 `byte` 类型，而是 `int` 类型呢？

因为 `read` 方法读到末尾时返回的是 `-1`，而在所操作的数据中的很容易出现连续多个 `1` 的情况，而连续读到 `8` 个 `1`，就是 `-1`，导致读取会提前停止。所以将读到的一个字节给提升为一个 `int` 类型的数值，但是只保留原字节，并在剩余二进制位补 `0`。

对于 **write** 方法，可以一次写入一个字节，但接收的是一个 **int** 类型数值。

只写入该 **int** 类型的数值的最低一个字节（**8** 位）。

简单说：**read** 方法对读到的数据进行提升。**write** 对操作的数据进行转换。这是神马意思？？？

转换流：

特点：

- 1，是字节流和字符流之间的桥梁。
- 2，该流对象中可以对读取到的字节数据进行指定编码表的编码转换。

什么时候使用呢？

- 1，当字节和字符之间有转换动作时。
- 2，流操作的数据需要进行编码表的指定时。

具体的对象体现：

1, `InputStreamReader`: 字节到字符的桥梁。

2, `OutputStreamWriter`: 字符到字节的桥梁。

这两个流对象是字符流体系中的成员。

那么它们有转换作用，而本身又是字符流。所以在构造的时候，需要传入字节流对象进来。

构造函数：

`InputStreamReader(InputStream)`:通过该构造函数初始化，使用的是本系统默认的编码表 GBK。

`InputStreamReader(InputStream,String charSet)`:通过该构造函数初始化，可以指定编码表。

`OutputStreamWriter(OutputStream)`:通过该构造函数初始化，使用的是本系统默认的编码表 GBK。

`OutputStreamWriter(OutputStream,String charSet)`:通过该构造函数初始化，可以指定编码表。

可以和流相关联的集合对象 `Properties`.

Map

|--Hashtable

|--Properties

`Properties`:该集合不需要泛型，因为该集合中的键值对都是 `String` 类型。

1, 存入键值对: `setProperty(key,value);`

2, 获取指定键对应的值: `value getProperty(key);`

3, 获取集合中所有键元素:

`Enumeration propertyNames();`

在 jdk1.6 版本给该类提供一个新的方法。

`Set<String> stringPropertyNames();`

4, 列出该集合中的所有键值对, 可以通过参数打印流指定列出到的目的地。

`list(PrintStream);`

`list(PrintWriter);`

例: `list(System.out)`:将集合中的键值对打印到控制台。

`list(new PrintStream("prop.txt"))`:将集合中的键值对存储到 `prop.txt` 文件中。

5, 可以将流中的规则数据加载进行集合, 并称为键值对。

`load(InputStream);`

jdk1.6 版本。提供了新的方法。

`load(Reader);`

注意: 流中的数据要是"键=值"的规则数据。

6, 可以将集合中的数据进行指定目的的存储。

`store(OutputStream,String comment)`方法。

jdk1.6 版本。提供了新的方法。

`store(Writer ,String comment)`:

使用该方法存储时, 会带着当时存储的时间。

File 类:

该类的出现是对文件系统的中的文件以及文件夹进行对象的封装。

可以通过对象的思想来操作文件以及文件夹。

1, 构造函数:

`File(String filename)`:将一个字符串路径(相对或者绝对)封装成 `File` 对象, 该路径是可存在的, 也可以是不存在。

`File(String parent,String child);`

`File(File parent,String child);`

2, 特别的字段: `separator`:跨平台的目录分隔符。

如: `File file = new File("c:"+File.separator+"abc"+File.separator+"a.txt");`

3, 常见方法:

1, 创建:

`boolean createNewFile()` throws `IOException`:创建文件, 如果被创建的文件已经存在, 则不创建。

`boolean mkdir()`: 创建文件夹。

`boolean mkdirs()`: 创建多级文件夹。

2, 删除:

`boolean delete()`:可用于删除文件或者文件夹。

注意: 对于文件夹只能删除不带内容的空文件夹,

对于带有内容的文件夹, 不可以直接删除, 必须要从里往外删除。

`void deleteOnExit()`: 删除动作交给系统完成。无论是否反生异常, 系统在退出时执行删除动作。

3, 判断:

`boolean canExecute()`:

`boolean canWrite()`:

`boolean canRead()`:

`boolean exists()`: 判断文件或者文件夹是否存在。

`boolean isFile()`: 判断 `File` 对象中封装的是否是文件。

`boolean isDirectory()`:判断 `File` 对象中封装的是否是文件夹。

`boolean isHidden()`:判断文件或者文件夹是否隐藏。在获取硬盘文件或者文件夹时,

对于系统目录中的文件，**java** 是无法访问的，所以在遍历，可以避免遍历隐藏文件。

4， 获取：

getName():获取文件或者文件夹的名称。

getPath():File 对象中封装的路径是什么，获取的就是什么。

getAbsolutePath():无论 File 对象中封装的路径是什么，获取的都是绝对路径。

getParent(): 获取 File 对象封装文件或者文件夹的父目录。

注意：如果封装的是相对路径，那么返回的是 **null**。

long length():获取文件大小。

long lastModified(): 获取文件或者文件最后一次修改的时间。

static File[] listRoots():获取的是被系统中有效的盘符。

String[] list():获取指定目录下当前的文件以及文件夹名称。

String[] list(Filenamefilter): 可以根据指定的过滤器，过滤后的文件及文件夹名称。

File[] listFiles():获取指定目录下的文件以及文件夹对象。

5， 重命名：

renameTo(File):

```
File f1 = new File("c:\\a.txt");
```

```
File f2 = new File("c:\\b.txt");
```

```
f1.renameTo(f2);//将 c 盘下的 a.txt 文件改名为 b.txt 文件。
```

对象的序列化。

ObjectInputStream

ObjectOutputStream

可以通过这两个流对象直接操作已有对象并将对象进行本地持久化存储。

存储后的对象可以进行网络传输。

Serializable: 该接口其实就是一个没有方法的标记接口。

用于给类指定一个 **UID**。该 **UID** 是通过类中的可序列化成员的数字签名运算出来的一个 **long** 型的值。

只要是这些成员没有变化，那么该值每次运算都一样。

该值用于判断被序列化的对象和类文件是否兼容。

如果被序列化的对象需要被不同的类版本所兼容。可以在类中自定义 **UID**。

定义方式：**static final long serialVersionUID = 42L;**

注意：对应静态的成员变量，不会被序列化。

对应非静态也不想被序列化的成员而言，可以通过 **transient** 关键字修饰。

通常，这两个对象成对使用。

其他的数据操作流

操作基本数据类型的流对象。

DataInputStream

DataInputStream(InputStream);

操作基本数据类型的方法：

int readInt();一次读取四个字节，并将其转成 **int** 值。

boolean readBoolean();一次读取一个字节。

short readShort();

long readLong();

剩下的数据类型一样。

`String readUTF()`:按照 utf-8 修改版读取字符。注意，它只能读 `writeUTF()`

写入的字符数据。

`DataOutputStream`

`DataOutputStream(OutputStream)`:

操作基本数据类型的方法:

`writeInt(int)`: 一次写入四个字节。

注意和 `write(int)`不同。`write(int)`只将该整数的最低一个 8 位写入。剩余三个 8 位丢弃。

`writeBoolean(boolean);`

`writeShort(short);`

`writeLong(long);`

剩下是数据类型也一样。

`writeUTF(String)`:按照 utf-8 修改版将字符数据进行存储。只能通过 `readUTF` 读取。

通常只要操作基本数据类型的数据。就需要通过 `DataStram` 进行包装。

通常成对使用。

操作数组的流对象。

1, 操作字节数组

`ByteArrayInputStream`

`ByteArrayOutputStream`

`toByteArray();`

`toString();`

`writeTo(OutputStream);`

2, 操作字符数组。

`CharArrayReader`

CharArrayWriter

对于这些流，源是内存。目的也是内存。

而且这些流并未调用系统资源。使用的就是内存中的数组。

所以这些在使用的时候不需要 **close**。

操作数组的读取流在构造时，必须要明确一个数据源。所以要传入相对应的数组。

对于操作数组的写入流，在构造函数可以使用空参数。因为它内置了一个可变长度数组作为缓冲区。

（七）——java 数据库操作

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

数据库访问几乎每一个稍微成型的程序都要用到的知识，怎么高效的访问数据库也是我们学习的一个重点，今天的任务就是总结 java 访问数据库的方法和有关 API，java 访问数据库主要用的方法是 JDBC,它是 java 语言中用来

规范客户端程序如何来访问数据库的应用程序接口，提供了诸如查询和更新数据库中数据的方法，下面我们就具体来总结一下 **JDBC**

一：Java 访问数据库的具体步骤：

1 加载（注册）数据库

驱动加载就是把各个数据库提供的访问数据库的 API 加载到我们程序进来，加载 JDBC 驱动，并将其注册到 DriverManager 中，每一种数据库提供的数据库驱动不一样，加载驱动时要把 jar 包添加到 lib 文件夹下，下面看一下一些主流数据库的 JDBC 驱动加载注册的代码：

//Oracle8/8i/9iO 数据库(thin 模式)

```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
```

//Sql Server7.0/2000 数据

```
库      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
```

//Sql Server2005/2008 数据

```
库      Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

//DB2 数据库

```
Class.froName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
```

//MySQL 数据

```
库      Class.forName("com.mysql.jdbc.Driver").newInstance();
```

2 建立链接

建立数据库之间的连接是访问数据库的必要条件，就像南水北调调水一样，要想调水首先由把沟通的河流打通。建立连接对于不同数据库也是不一样的，

下面看一下一些主流数据库建立数据库连接，取得 Connection 对象的不同方式：

//Oracle8/8i/9i 数据库(thin 模式)

```
String url="jdbc:oracle:thin:@localhost:1521:orcl";
String user="scott";
String password="tiger";
Connection conn=DriverManager.getConnection(url,user,password);
```

//Sql Server7.0/2000/2005/2008 数据库

```
String url="jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=pubs";
String user="sa";
String password="";
Connection conn=DriverManager.getConnection(url,user,password);
```

//DB2 数据库

```
String url="jdbc:db2://localhost:5000/sample";
String user="amdin"
String password="-";
Connection conn=DriverManager.getConnection(url,user,password);
```

//MySQL 数据库

```
String url="jdbc:mysql://localhost:3306/testDB?user=root&password=root&useUnicode=true&characterEncoding=gb2312";
Connection conn=DriverManager.getConnection(url);
```

3. 执行 SQL 语句

数据库连接建立好之后，接下来就是一些准备工作和执行 sql 语句了，准备工作要做的就是建立 Statement 对象 PreparedStatement 对象，例如：

//建立 Statement 对象

```
Statement stmt=conn.createStatement();
```

//建立 PreparedStatement 对象

```
String sql="select * from user where userName=? and password=?";
```

```
PreparedStatement pstmt=Conn.prepareStatement(sql);
```

```
pstmt.setString(1,"admin");
```

```
pstmt.setString(2,"liubin");
```

做好准备工作之后就可以执行 sql 语句了，执行 sql 语句：

```
String sql="select * from users";
```

```
ResultSet rs=stmt.executeQuery(sql);
```

//执行动态 SQL 查询

```
ResultSet rs=pstmt.executeQuery();
```

//执行 insert update delete 等语句，先定义 sql

```
stmt.executeUpdate(sql);
```

4 处理结果集

访问结果记录集 ResultSet 对象。例如：

```
while(rs.next)
```

```
{
```

```
out.println("你的第一个字段内容为: "+rs.getString("Name"));
```

```
out.println("你的第二个字段内容为: "+rs.getString(2));
```

```
}
```

5 关闭数据库

依次将 ResultSet、Statement、PreparedStatement、Connection 对象

关闭，释放所占用的资源.例如：

```
rs.close();
```

```
stmt.close();
```

```
pstmt.close();
```

```
con.close();
```

二：JDBC 事务

什么是事务:

首先,说说什么事务。我认为事务,就是一组操作数据库的动作集合。

事务是现代数据库理论中的核心概念之一。如果一组处理步骤或者全部发生或者一步也不执行,我们称该组处理步骤为一个事务。当所有的步骤像一个操作一样被完整地执行,我们称该事务被提交。由于其中的一部分或多步执行失败,导致没有步骤被提交,则事务必须回滚到最初的系统状态。

事务必须服从 ISO/IEC 所制定的 ACID 原则。ACID 是原子性 (atomicity)、一致性 (consistency)、隔离性 (isolation) 和持久性 (durability) 的缩写。事务的原子性表示事务执行过程中的任何失败都将导致事务所做的任何修改失效。一致性表示 当事务执行失败时,所有被该事务影响的数据都应该恢复到事务执行前的状态。隔离性表示在事务执行过程中对数据的修改,在事务提交之前对其他事务不可见。持久性表示当系统或介质发生故障时,确保已提交事务的更新不能丢失。持久性通过数据库备份和恢复来保证。

JDBC 事务是用 Connection 对象控制的。JDBC Connection 接口

(`java.sql.Connection`)提供了两种事务模式:自动提交和手工提

交。`java.sql.Connection` 提供了以下控制事务的方法:

```
public void setAutoCommit(boolean)
public boolean getAutoCommit()
public void commit()
public void rollback()
```

使用 JDBC 事务界定时,您可以将多个 SQL 语句结合到一个事务中。

JDBC 事务的一个缺点是事务的范围局限于一个数据库连接。一

个 JDBC 事务不能跨越多个数据库。

三: java 操作数据库连接池

在总结 java 操作数据库连接池发现一篇很好的文章，所以就不做具体总结了，直接上地址：

<http://www.blogjava.net/chunkyo/archive/2007/01/16/94266.html>

最后附一段比较经典的代码吧：

[java] [view plaincopyprint?](#)

```
1. import java.sql.Connection;
2. import java.sql.DatabaseMetaData;
3. import java.sql.Driver;
4. import java.sql.DriverManager;
5. import java.sql.SQLException;
6. import java.sql.Statement;
7. import java.util.Enumeration;
8. import java.util.Vector;
9. public class ConnectionPool {
10. private String jdbcDriver = ""; // 数据库驱动
11. private String dbUrl = ""; // 数据 URL
12. private String dbUsername = ""; // 数据库用户名
13. private String dbPassword = ""; // 数据库用户密码
14. private String testTable = ""; // 测试连接是否可用的测试表名，默认没有测试表
15. private int initialConnections = 10; // 连接池的初始大小
16. private int incrementalConnections = 5; // 连接池自动增加的大小
17. private int maxConnections = 50; // 连接池最大的大小
18. private Vector connections = null; // 存放连接池中数据库连接的向量，初始时
    为 null
19.
20. // 它中存放的对象为 PooledConnection 型
21.
22. /**
23. * 构造函数
24. *
25. * @param jdbcDriver String JDBC 驱动类串
26. * @param dbUrl String 数据库 URL
27. * @param dbUsername String 连接数据库用户名
28. * @param dbPassword String 连接数据库用户的密码
29. *
30. */
```



```

31.
32. public ConnectionPool(String jdbcDriver,String dbUrl,String dbUsername,String db
    Password) {
33.     this.jdbcDriver = jdbcDriver;
34.     this.dbUrl = dbUrl;
35.     this.dbUsername = dbUsername;
36.     this.dbPassword = dbPassword;
37. }
38.
39. /**
40.
41. * 返回连接池的初始大小
42. *
43. * @return 初始连接池中可获得的连接数量
44. */
45. public int getInitialConnections() {
46.
47.     return this.initialConnections;
48. }
49.
50. /**
51.
52. * 设置连接池的初始大小
53.
54. *
55.
56. * @param 用于设置初始连接池中连接的数量
57.
58. */
59.
60. public void setInitialConnections(int initialConnections) {
61.     this.initialConnections = initialConnections;
62. }
63.
64. /**

```

```
65.
66.* 返回连接池自动增加的大小、
67.*
68.* @return 连接池自动增加的大小
69.*/
70.public int getIncrementalConnections() {
71.
72.    return this.incrementalConnections;
73.
74.}
75.
76./**
77.* 设置连接池自动增加的大小
78.* @param 连接池自动增加的大小
79.*/
80.
81.public void setIncrementalConnections(int incrementalConnections) {
82.
83.    this.incrementalConnections = incrementalConnections;
84.
85.}
86.
87./**
88.* 返回连接池中最大的可用连接数量
89.* @return 连接池中最大的可用连接数量
90.*/
91.
92.public int getMaxConnections() {
93.    return this.maxConnections;
94.}
95.
96./**
97.
98.* 设置连接池中最大可用的连接数量
99.
```

```

100.  *
101.
102.  * @param 设置连接池中最大可用的连接数量值
103.
104.  */
105.
106.  public void setMaxConnections(int maxConnections) {
107.
108.      this.maxConnections = maxConnections;
109.
110.  }
111.
112.  /**
113.
114.  * 获取测试数据库表的名字
115.  *
116.  * @return 测试数据库表的名字
117.  */
118.  public String getTestTable() {
119.
120.      return this.testTable;
121.
122.  }
123.
124.  /**
125.  * 设置测试表的名字
126.  * @param testTable String 测试表的名字
127.  */
128.  public void setTestTable(String testTable) {
129.      this.testTable = testTable;
130.  }
131.
132.  /**
133.
134.  *

```

```

135.  * 创建一个数据库连接池，连接池中的可用连接的数量采用类成员
136.  * initialConnections 中设置的值
137.  */
138.  public synchronized void createPool() throws Exception {
139.
140.      // 确保连接池没有创建
141.
142.      // 如果连接池已经创建了，保存连接的向量 connections 不会为空
143.
144.      if (connections != null) {
145.
146.          return; // 如果已经创建，则返回
147.
148.      }
149.
150.      // 实例化 JDBC Driver 中指定的驱动类实例
151.
152.      Driver driver = (Driver) (Class.forName(this.jdbcDriver).newInstance());
153.
154.      DriverManager.registerDriver(driver); // 注册 JDBC 驱动程序
155.
156.      // 创建保存连接的向量，初始时有 0 个元素
157.
158.      connections = new Vector();
159.
160.      // 根据 initialConnections 中设置的值，创建连接。
161.
162.      createConnections(this.initialConnections);
163.
164.      System.out.println(" 数据库连接池创建成功！ ");
165.
166.  }
167.
168.  /**
169.

```

```

170.  * 创建由 numConnections 指定数目的数据库连接，并把这些连接
171.
172.  * 放入 connections 向量中
173.  *
174.  * @param numConnections 要创建的数据库连接的数目
175.  */
176.  @SuppressWarnings("unchecked")
177.  private void createConnections(int numConnections) throws SQLException {
178.
179.      // 循环创建指定数目的数据库连接
180.
181.      for (int x = 0; x < numConnections; x++) {
182.
183.          // 是否连接池中的数据库连接的数量已经达到最大？最大值由类成
            员 maxConnections
184.
185.          // 指出，如果 maxConnections 为 0 或负数，表示连接数量没有限制。
186.
187.          // 如果连接数已经达到最大，即退出。
188.
189.          if (this.maxConnections > 0 && this.connections.size() >= this.maxConn
            ections) {
190.
191.              break;
192.
193.          }
194.
195.          //add a new PooledConnection object to connections vector
196.
197.          // 增加一个连接到连接池中（向量 connections 中）
198.
199.          try{
200.
201.              connections.addElement(new PooledConnection(newConnection()));
202.

```

```

203.         }catch(SQLException e){
204.
205.             System.out.println(" 创建数据库连接失败！ "+e.getMessage());
206.
207.             throw new SQLException();
208.
209.         }
210.
211.         System.out.println(" 数据库连接已创建 .....");
212.
213.     }
214. }
215.
216. /**
217.
218.  * 创建一个新的数据库连接并返回它
219.  *
220.  * @return 返回一个新创建的数据库连接
221.  */
222. private Connection newConnection() throws SQLException {
223.
224.     // 创建一个数据库连接
225.
226.     Connection conn = DriverManager.getConnection(dbUrl, dbUsername,
        dbPassword);
227.
228.     // 如果这是第一次创建数据库连接，即检查数据库，获得此数据库允许支
        持的
229.
230.     // 最大客户连接数目
231.
232.     //connections.size()==0 表示目前没有连接已被创建
233.
234.     if (connections.size() == 0) {
235.

```

```

236.         DatabaseMetaData metaData = conn.getMetaData();
237.
238.         int driverMaxConnections = metaData.getMaxConnections();
239.
240.         // 数据库返回的 driverMaxConnections 若为 0，表示此数据库没有最
           大
241.
242.         // 连接限制，或数据库的最大连接限制不知道
243.
244.         //driverMaxConnections 为返回的一个整数，表示此数据库允许客户连接
           的数目
245.
246.         // 如果连接池中设置的最大连接数量大于数据库允许的连接数目，则置连
           接池的最大
247.
248.         // 连接数目为数据库允许的最大数目
249.
250.         if (driverMaxConnections > 0 && this.maxConnections > driverMaxCon
           nections) {
251.
252.             this.maxConnections = driverMaxConnections;
253.
254.         }
255.     }
256.     return conn; // 返回创建的新的数据库连接
257.
258. }
259.
260. /**
261.
262. * 通过调用 getFreeConnection() 函数返回一个可用的数据库连接，
263.
264. * 如果当前没有可用的数据库连接，并且更多的数据库连接不能创
265.
266. * 建（如连接池大小的限制），此函数等待一会再尝试获取。

```

```

267.
268.  *
269.
270.  * @return 返回一个可用的数据库连接对象
271.
272.  */
273.
274.  public synchronized Connection getConnection() throws SQLException {
275.
276.      // 确保连接池已被创建
277.
278.      if (connections == null) {
279.
280.          return null; // 连接池还没创建，则返回 null
281.
282.      }
283.
284.      Connection conn = getFreeConnection(); // 获得一个可用的数据库连
      接
285.
286.      // 如果目前没有可以使用的连接，即所有的连接都在使用中
287.
288.      while (conn == null){
289.
290.          // 等一会再试
291.
292.          wait(250);
293.
294.          conn = getFreeConnection(); // 重新再试，直到获得可用的连接，如果
295.
296.          //getFreeConnection() 返回的为 null
297.
298.          // 则表明创建一批连接后也不可获得可用连接
299.
300.      }

```



```

301.
302.         return conn;// 返回获得的可用的连接
303.     }
304.
305.     /**
306.
307.     * 本函数从连接池向量 connections 中返回一个可用的的数据库连接，如果
308.
309.     * 当前没有可用的数据库连接，本函数则根据 incrementalConnections 设置
310.
311.     * 的值创建几个数据库连接，并放入连接池中。
312.
313.     * 如果创建后，所有的连接仍都在使用中，则返回 null
314.
315.     * @return 返回一个可用的数据库连接
316.
317.     */
318.
319.     private Connection getFreeConnection() throws SQLException {
320.
321.         // 从连接池中获得一个可用的数据库连接
322.
323.         Connection conn = findFreeConnection();
324.
325.         if (conn == null) {
326.
327.             // 如果目前连接池中没有可用的连接
328.
329.             // 创建一些连接
330.
331.             createConnections(incrementalConnections);
332.
333.             // 重新从池中查找是否有可用连接
334.
335.             conn = findFreeConnection();

```

```

336.
337.         if (conn == null) {
338.
339.             // 如果创建连接后仍获得不到可用的连接，则返回 null
340.
341.             return null;
342.
343.         }
344.
345.     }
346.
347.     return conn;
348.
349. }
350.
351. /**
352.
353. * 查找连接池中所有的连接，查找一个可用的数据库连接，
354.
355. * 如果没有可用的连接，返回 null
356.
357. *
358.
359. * @return 返回一个可用的数据库连接
360.
361. */
362.
363. private Connection findFreeConnection() throws SQLException {
364.
365.     Connection conn = null;
366.
367.     PooledConnection pConn = null;
368.
369.     // 获得连接池向量中所有的对象
370.

```

```
371.      Enumeration enumerate = connections.elements();
372.
373.      // 遍历所有的对象，看是否有可用的连接
374.
375.      while (enumerate.hasMoreElements()) {
376.
377.          pConn = (PooledConnection) enumerate.nextElement();
378.
379.          if (!pConn.isBusy()) {
380.
381.              // 如果此对象不忙，则获得它的数据库连接并把它设为忙
382.
383.              conn = pConn.getConnection();
384.
385.              pConn.setBusy(true);
386.
387.              // 测试此连接是否可用
388.
389.              if (!testConnection(conn)) {
390.
391.                  // 如果此连接不可再用了，则创建一个新的连接，
392.
393.                  // 并替换此不可用的连接对象，如果创建失败，返回 null
394.
395.                  try{
396.
397.                      conn = newConnection();
398.
399.                  }catch(SQLException e){
400.
401.                      System.out.println(" 创建数据库连接失败！ "+e.getMessage());
402.
403.                      return null;
404.
405.                  }
```

```

406.
407.         pConn.setConnection(conn);
408.
409.     }
410.
411.         break; // 已经找到一个可用的连接，退出
412.
413.     }
414.
415. }
416.
417.     return conn; // 返回找到的可用连接
418.
419. }
420.
421. /**
422.
423. * 测试一个连接是否可用，如果不可用，关掉它并返回 false
424.
425. * 否则可用返回 true
426.
427. *
428.
429. * @param conn 需要测试的数据库连接
430.
431. * @return 返回 true 表示此连接可用， false 表示不可用
432.
433. */
434.
435. private boolean testConnection(Connection conn) {
436.
437.     try {
438.
439.         // 判断测试表是否存在
440.

```

```

441.         if (testTable.equals("")) {
442.
443.             // 如果测试表为空，试着使用此连接的 setAutoCommit() 方法
444.
445.             // 来判断连接否可用（此方法只在部分数据库可用，如果不可用，
446.
447.             // 抛出异常）。注意：使用测试表的方法更可靠
448.
449.             conn.setAutoCommit(true);
450.
451.         } else { // 有测试表的时候使用测试表测试
452.
453.             //check if this connection is valid
454.
455.             Statement stmt = conn.createStatement();
456.
457.             stmt.execute("select count(*) from " + testTable);
458.
459.         }
460.
461.     } catch (SQLException e) {
462.
463.         // 上面抛出异常，此连接已不可用，关闭它，并返回 false;
464.
465.         closeConnection(conn);
466.
467.         return false;
468.
469.     }
470.
471.     // 连接可用，返回 true
472.
473.     return true;
474.
475. }

```

```

476.
477.  /**
478.
479.  * 此函数返回一个数据库连接到连接池中，并把此连接置为空闲。
480.
481.  * 所有使用连接池获得的数据库连接均应在不使用此连接时返回它。
482.
483.  *
484.
485.  * @param 需返回到连接池中的连接对象
486.
487.  */
488.
489. public void returnConnection(Connection conn) {
490.
491.     // 确保连接池存在，如果连接没有创建（不存在），直接返回
492.
493.     if (connections == null) {
494.
495.         System.out.println(" 连接池不存在，无法返回此连接到连接池中 !");
496.
497.         return;
498.
499.     }
500.
501.     PooledConnection pConn = null;
502.
503.     Enumeration enumerate = connections.elements();
504.
505.     // 遍历连接池中的所有连接，找到这个要返回的连接对象
506.
507.     while (enumerate.hasMoreElements()) {
508.
509.         pConn = (PooledConnection) enumerate.nextElement();
510.

```

```

511.         // 先找到连接池中的要返回的连接对象
512.
513.         if (conn == pConn.getConnection()) {
514.
515.             // 找到了，设置此连接为空闲状态
516.
517.             pConn.setBusy(false);
518.
519.             break;
520.
521.         }
522.
523.     }
524.
525. }
526.
527. /**
528.
529.  * 刷新连接池中所有的连接对象
530.
531.  *
532.
533.  */
534.
535. public synchronized void refreshConnections() throws SQLException {
536.
537.     // 确保连接池已创新存在
538.
539.     if (connections == null) {
540.
541.         System.out.println(" 连接池不存在，无法刷新 !");
542.
543.         return;
544.
545.     }

```

```

546.
547.     PooledConnection pConn = null;
548.
549.     Enumeration enumerate = connections.elements();
550.
551.     while (enumerate.hasMoreElements()) {
552.
553.         // 获得一个连接对象
554.
555.         pConn = (PooledConnection) enumerate.nextElement();
556.
557.         // 如果对象忙则等 5 秒 ,5 秒后直接刷新
558.
559.         if (pConn.isBusy()) {
560.
561.             wait(5000); // 等 5 秒
562.
563.         }
564.
565.         // 关闭此连接，用一个新的连接代替它。
566.
567.         closeConnection(pConn.getConnection());
568.
569.         pConn.setConnection(newConnection());
570.
571.         pConn.setBusy(false);
572.
573.     }
574.
575. }
576.
577. /**
578.
579. * 关闭连接池中所有的连接，并清空连接池。
580.

```



```

581.  */
582.
583.  public synchronized void closeConnectionPool() throws SQLException {
584.
585.      // 确保连接池存在，如果不存在，返回
586.
587.      if (connections == null) {
588.
589.          System.out.println(" 连接池不存在，无法关闭 !");
590.
591.          return;
592.
593.      }
594.
595.      PooledConnection pConn = null;
596.
597.      Enumeration enumerate = connections.elements();
598.
599.      while (enumerate.hasMoreElements()) {
600.
601.          pConn = (PooledConnection) enumerate.nextElement();
602.
603.          // 如果忙，等 5 秒
604.
605.          if (pConn.isBusy()) {
606.
607.              wait(5000); // 等 5 秒
608.
609.          }
610.
611.          //5 秒后直接关闭它
612.
613.          closeConnection(pConn.getConnection());
614.
615.          // 从连接池向量中删除它

```

```
616.
617.         connections.removeElement(pConn);
618.
619.     }
620.
621.     // 置连接池为空
622.
623.     connections = null;
624.
625. }
626.
627. /**
628.
629.  * 关闭一个数据库连接
630.
631.  *
632.
633.  * @param 需要关闭的数据库连接
634.
635.  */
636.
637. private void closeConnection(Connection conn) {
638.
639.     try {
640.
641.         conn.close();
642.
643.     }catch (SQLException e) {
644.
645.         System.out.println(" 关闭数据库连接出错: "+e.getMessage());
646.
647.     }
648.
649. }
650.
```

```

651.  /**
652.
653.  * 使程序等待给定的毫秒数
654.
655.  *
656.
657.  * @param 给定的毫秒数
658.
659.  */
660.
661.  private void wait(int mSeconds) {
662.
663.      try {
664.
665.          Thread.sleep(mSeconds);
666.
667.      } catch (InterruptedException e) {
668.
669.      }
670.
671.  }
672.
673.  /**
674.
675.  *
676.
677.  * 内部使用的用于保存连接池中连接对象的类
678.
679.  * 此类中有两个成员，一个是数据库的连接，另一个是指示此连接是否
680.
681.  * 正在使用的标志。
682.
683.  */
684.
685.  class PooledConnection {

```

```

686.
687.     Connection connection = null;// 数据库连接
688.
689.     boolean busy = false; // 此连接是否正在使用的标志，默认没有正在使
        用
690.
691.     // 构造函数，根据一个 Connection 构造一个 PooledConnection 对象
692.
693.     public PooledConnection(Connection connection) {
694.
695.         this.connection = connection;
696.
697.     }
698.
699.     // 返回此对象中的连接
700.
701.     public Connection getConnection() {
702.
703.         return connection;
704.
705.     }
706.
707.     // 设置此对象的，连接
708.
709.     public void setConnection(Connection connection) {
710.
711.         this.connection = connection;
712.
713.     }
714.
715.     // 获得对象连接是否忙
716.
717.     public boolean isBusy() {
718.
719.         return busy;

```

```

720.
721.     }
722.
723.     // 设置对象的连接正在忙
724.
725.     public void setBusy(boolean busy) {
726.
727.         this.busy = busy;
728.
729.     }
730.
731. }
732.
733. }
734.
735. =====
736.
737. 这个例子是根据 POSTGRESQL 数据库写的，
738. 请用的时候根据实际的数据库调整。
739.
740. 调用方法如下：
741.
742. ① ConnectionPool connPool
743.         = new ConnectionPool("org.postgresql.Driver"
744.                                ,"jdbc:postgresql://dbURI:5432/
       DBName"
                                ,"postgre"
                                ,"postgre");
745.
746.
747.
748. ② connPool .createPool();
749.     Connection conn = connPool .getConnection();

```

（八）反射和代理机制

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

反射和代理机制是 JDK5.0 提供的 java 新特性，反射的出现打破了 java 一些常规的规则，如，私有变量不可访问。但反射和代理在学习过程中也是一个比较难理解的知识点。本人曾经学过一段时间的反射和代理，但好长时间不用好像有点生疏了，当时学的时候就理解的不是很透彻，这次总结算是重新学习一遍吧，如果有什么错误，请大家拍砖：

先看一下，**Java** 反射机制主要提供了以下功能：

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法

一般而言，开发者社群说到动态语言，大致认同的一个定义是：“程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言”尽管在这样的定义与分类下 **Java** 不是动态语言，它却有着一个非常突出的动态相关机制：**Reflection**。这个字的意思是“反射、映象、倒影”，用在 **Java** 身上指的是我们可以于运行时加载、探知、使用编译期间完全未知的 **classes**。换句话说，**Java** 程序可以加载一个运行时才得知名称的 **class**，获悉其完整构造（但不包括 **methods** 定义），并生成其对象实体、或对其 **fields** 设值、或唤起其 **methods**。这种“看透 **class**”的能力

（the ability of the program to examine itself）被称为 introspection（内省、内观、反省）。Reflection 和 introspection 是常被并提的两个术语

API 简介

在 JDK 中，主要由以下类来实现 Java 反射机制，这些类都位于 `java.lang.reflect` 包中

- Class 类：代表一个类。
- Field 类：代表类的成员变量（成员变量也称为类的属性）。
- Method 类：代表类的方法。
- Constructor 类：代表类的构造方法。
- Array 类：提供了动态创建数组，以及访问数组的元素的静态方法

在 `java.lang.Object` 类中定义了 `getClass()` 方法，因此对于任意一个 Java 对象，都可以通过此方法获得对象的类型。

Class 类是 Reflection API 中的核心类，它有以下方法

- `getName()`：获得类的完整名字。
- `getFields()`：获得类的 `public` 类型的属性。
- `getDeclaredFields()`：获得类的所有属性。
- `getMethods()`：获得类的 `public` 类型的方法。
- `getDeclaredMethods()`：获得类的所有方法。
- `getMethod(String name, Class[] parameterTypes)`：获得类的特定方法，`name` 参数指定方法的名字，`parameterTypes` 参数指定方法的参数类型。
- `getConstructors()`：获得类的 `public` 类型的构造方法。

- `getConstructor(Class[] parameterTypes)`: 获得类的特定构造方法,
`parameterTypes` 参数指定构造方法的参数类型。
- `newInstance()`: 通过类的不带参数的构造方法创建这个类的一个对象。
- (2) 通过默认构造方法创建一个新对象:
- `Object objectCopy=classType.getConstructor(new Class[]{}).newInstance(new Object[]{});`
- 以上代码先调用 `Class` 类的 `getConstructor()`方法获得一个 `Constructor` 对象, 它代表默认的构造方法, 然后调用 `Constructor` 对象的 `newInstance()`方法构造一个实例。

- (3) 获得对象的所有属性:

- `Field fields[]=classType.getDeclaredFields();`
- `Class` 类的 `getDeclaredFields()`方法返回类的所有属性, 包括 `public`、`protected`、默认和 `private` 访问级别的属性

(4) `Method` 类的 `invoke(Object obj, Object args[])`方法接收的参数必须为对象, 如果参数为基本类型数据, 必须转换为相应的包装类型的对象。`invoke()`方法的返回值总是对象, 如果实际被调用的方法的返回类型是基本类型数据, 那么 `invoke()`方法会把它转换为相应的包装类型的对象, 再将其返回

(5) `Java` 中, 无论生成某个类的多少个对象, 这些对象都会对应于同一个 `Class` 对象。要想使用反射, 首先需要获得待处理类或对象所对应的 `Class` 对象。

获取某个类或某个对象所对应的 `Class` 对象的常用的 3 种方式:

a) 使用 `Class` 类的静态方法 `forName`:

`Class.forName("java.lang.String");`

b) 使用类的 `.class` 语法: `String.class`;

c) 使用对象的 getClass()方法:

String s = "aa"; Class<?> clazz = s.getClass();

下面写一个程序来用一下这些 API 吧:

[java] [view plaincopyprint?](#)

```
1. //获得 MethodInvoke 类对应的一个 class 对象
2. Class<?> MethodInvok=MethodInvoke.class;
3. //获得一个 MethodInvoke 类对应的对象实例
4. Object MethodInvo=MethodInvok.newInstance();
5. //获得 MethodInvo 对象对应的 add 方法对应的一个对象实例
6. 1) : Method method=MethodInvok.getMethod("add", int.class,int.class);
7. //调用 MethodInvo 对象对应的 add 方法对应的一个对象(MethodInvo)实例所代表的
   方法, 并获得结果
8.      2) Object result= method.invoke(MethodInvo, 1,2);
9.      System.out.println(result);
10.     System.out.println("-----");
11.     Method method1=MethodInvok.getMethod("print",String.class);
12.     Object Result1=method1.invoke(MethodInvo, "tom");
13.     System.out.println(Result1);
```

注: 1) 处的 int.class,int.class 可以写为 new Class[]{int.class,int.class}

原因在于 getMethod 方法的第二个参数是一个可变参数。

2) 处的 1,2 可以写为 new int【】 { 1,2}, 原因如 1) ;

4.若想通过类的不带参数的构造方法来生成对象, 我们有两种方式:

a) 先获得 Class 对象, 然后通过该 Class 对象的 newInstance()方法直接生成即可:

```
Class<?> classType = String.class;
Object obj = classType.newInstance();
```

b) 先获得 Class 对象, 然后通过该对象获得对应的 Constructor 对象, 再通过该 Constructor 对象的 newInstance()方法生成:

```
Class<?> classType = Customer.class;
Constructor cons = classType.getConstructor(new Class[]{});
Object obj = cons.newInstance(new Object[]{});
```

注：

4. 若想通过类的带参数的构造方法生成对象，只能使用下面这一种方式：

```
Class<?> classType = Customer.class;
Constructor cons = classType.getConstructor(new Class[]{String.class, int.class});
Object obj = cons.newInstance(new Object[]{"hello", 3});
```

代码示例：

[java] [view plaincopyprint?](#)

```
1.    // 该方法实现对 Customer 对象的拷贝操作
2.    public Object copy(Object object) throws Exception
3.    {
4.        Class<?> classType = object.getClass();
5.        //先调用 Class 类的 getConstructor()方法获得一个 Constructor 对象，它代表默
        认的构造方法，然后调用
6.        Constructor 对象的 newInstance()方法构造一个实例。
7.        Object objectCopy = classType.getConstructor(new Class[] {})
8.            .newInstance(new Object[] {});
9.        // Class 类的 getDeclaredFields()方法返回类的所有属性，包括 public、protected、
        默
10.       认和 private 访问级别的属性
11.        Field[] fields = classType.getDeclaredFields();
12.        for (Field field : fields)
13.        {
14.            String name = field.getName();
15.            // 将属性的首字母转换为大写
16.            String firstLetter = name.substring(0, 1).toUpperCase();      String getMet
            hodName = "get" + firstLetter + name.substring(1);
17.            String setMethodName = "set" + firstLetter + name.substring(1);
18.            Method getMethod = classType.getMethod(getMethodName,
19.                new Class[] {});
```

```

20.    Method setMethod = classType.getMethod(setMethodName,
21.        new Class[] { field.getType() });
22.    Object value = getMethod.invoke(object, new Object[] {});
23.    setMethod.invoke(objectCopy, new Object[] { value });
24. }
25. // 以上两行代码等价于下面一行
26. // Object obj2 = classType.newInstance();
27. // System.out.println(obj);
28. return objectCopy;
29.}

```

5.Integer.TYPE 返回的是 int，而 Integer.class 返回的是 Integer 类所对应的 Class 对象。

java.lang.Array 类提供了动态创建和访问数组元素的各种静态方法

一维数组的简单创建，设值，取值

```
Object array = Array.newInstance(classType, 10);
```

```
Array.set(array, 5, "hello");
```

```
String str = (String)Array.get(array, 5);
```

二维数组的简单创建，设值，取值

[java] [view plaincopyprint?](#)

```

1.    //创建一个设值数组维度的数组
2.    int[] dims = new int[] { 5, 10, 15 };
3.    //利用 Array.newInstance 创建一个数组对象，第一个参数指定数组的类型，第
4.    二个参数设值数组的维度，下面是创建一个长宽高为：5,10,15 的三维数组
5. Object array = Array.newInstance(Integer.TYPE, dims);
6.
7. System.out.println(array instanceof int[][][]);
8.    //获取三维数组的索引为 3 的一个二维数组
9. Object arrayObj = Array.get(array, 3);
10.   //获取二维数组的索引为 5 的一个一维数组

```

```

11.arrayObj = Array.get(arrayObj, 5);
12. //设值一维数组 arrayObj 下标为 10 的值设为 37
13.Array.setInt(arrayObj, 10, 37);
14.
15.int[][][] arrayCast = (int[][][]) array;
16.
17.System.out.println(arrayCast[3][5][10]);

```

利用反射访问类的私有方法：

代码示例：

[java] view plaincopyprint?

```

1. Private p = new Private();
2.
3. Class<?> classType = p.getClass();
4.
5. Method method = classType.getDeclaredMethod("sayHello",
6. new Class[] { String.class });
7. method.setAccessible(true);//压制 Java 的访问控制检查,使允许访问 private 方法
8. String str = (String)method.invoke(p, new Object[]{"zhangsan"});
9. System.out.println(str);

```

利用反射访问类的私有变量：

[java] view plaincopyprint?

```

1. Private2 p = new Private2();
2.
3. Class<?> classType = p.getClass();
4.
5. Field field = classType.getDeclaredField("name");
6.

```

```
7. field.setAccessible(true);//压制 Java 对访问修饰符的检查
8.
9. field.set(p, "lisi");
10.
11. System.out.println(p.getName());
```

代理

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。

- 在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用

代理模式一般涉及到的角色有

- 抽象角色：声明真实对象和代理对象的共同接口

- 代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装

- 真实角色：代理角色所代表的真实对象，是我们最终要引用的对象

Java 动态代理类位于 `java.lang.reflect` 包下，一般主要涉及到以下两个类：

- (1)Interface `InvocationHandler`：该接口中仅定义了一个方法

- `public Object invoke(Object obj, Method method, Object[] args)`

- 在实际使用时，第一个参数 `obj` 一般是指代理类，`method` 是被代理的方法，如上例中的 `request()`，`args` 为该方法的参数数组。这个抽象方法在代理类中动态实现。

•(2)Proxy: 该类即为动态代理类, 作用类似于上例中的 ProxySubject, 其中主要包含以下内容

•protected Proxy(InvocationHandler h): 构造函数, 用于给内部的 h 赋值。

•static Class getProxyClass (ClassLoader loader, Class[] interfaces): 获得一个代理类, 其中 loader 是类装载器, interfaces 是真实类所拥有的全部接口的数组。

•static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h): 返回代理类的一个实例, 返回后的代理类可以当作被代理类使用(可使用被代理类的在 Subject 接口中声明过的方法)

所谓 Dynamic Proxy (动态代理) 是这样一种 class: 它是在运行时生成的 class, 在生成它时你必须提供一组 interface 给它, 然后该 class 就宣称它实现了这些 interface。你当然可以把该 class 的实例当作这些 interface 中的任何一个来用。当然, 这个 Dynamic Proxy 其实就是一个 Proxy, 它不会替你作实质性的工作, 在生成它的实例时你必须提供一个 handler, 由它接管实际的工作

在使用动态代理类时, 我们必须实现 InvocationHandler 接口

通过代理的方式, 被代理的对象(RealSubject)可以在运行时动态改变, 需要控制的接口(Subject 接口)可以在运行时改变, 控制的方式(DynamicSubject 类)也可以动态改变, 从而实现了非常灵活的动态代理关系

动态代理是指客户通过代理类来调用其它对象的方法

•动态代理使用场合:

•调试

- 远程方法调用(RMI)

动态代理的步骤:

- 1.创建一个实现接口 **InvocationHandler** 的类，它必须实现 **invoke** 方法
- 2.创建被代理的类以及接口
- 3.通过 **Proxy** 的静态方法
**newProxyInstance(ClassLoader loader, Class[] interfaces, Invocation
Handler h)** 创建一个代理
- 4.通过代理调用方法

(九) ——数据库有关知识补充（事务、视图、索引、存储过程）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

一：事务

首先看一下什么是事务：

通俗的理解，事务是一组原子操作单元，从数据库角度说，就是一组 SQL 指令，要么全部执行成功，若因为某个原因其中一条指令执行有错误，则撤销先前执行过的所有指令。更简答的说就是：要么全部执行成功，要么撤销不执行。

然后看一下事务要遵循的 **ISO/IEC** 所制定的 **ACID** 原则：

ACID 是原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）的缩写。

1.事务的原子性表示事务执行过程中的任何失败都将导致事务所做的任何修改失效。

2.一致性表示当事务执行失败时，所有被该事务影响的数据都应该恢复到事务执行前的状态。

3.隔离性表示在事务执行过程中对数据的修改，在事务提交之前对其他事务不可见。

4.持久性表示已提交的数据在事务执行失败时，数据的状态都应该正确。

看一下一些准备知识：

1.T-SQL 使用下列语句来管理事务：

开始事务：BEGIN TRANSACTION

提交事务：COMMIT TRANSACTION

回滚（撤销）事务：ROLLBACK TRANSACTION

一旦事务提交或回滚，则事务结束。

2.判断某条语句执行是否出错：

使用全局变量@@ERROR

@@ERROR 只能判断当前一条 T-SQL 语句执行是否有错，为了判断事务中所有 T-SQL 语句是否有错，我们需要对错误进行累计

如：SET @errorSum=@errorSum+@@error

了解一下事务的分类：

显示事务：用 BEGIN TRANSACTION 明确指定事务的开始，这是最常用的事务类型

隐性事务：通过设置 SET IMPLICIT_TRANSACTIONS ON 语句，将隐性事务模式设置为打开，下一个语句自动启动一个新事务。当该事务完成时，再下一个 T-SQL 语句又将启动一个新事务

自动提交事务：这是 SQL Server 的默认模式，它将每条单独的 T-SQL 语句视为一个事务，如果成功执行，则自动提交；如果错误，则自动回滚

使用事务解决经典银行转账事务问题

T-SQL 语句：

[sql] [view plaincopyprint?](#)

```
1. BEGIN TRANSACTION
2. /*--定义变量，用于累计事务执行过程中的错误--*/
3. DECLARE @errorSum INT
4. SET @errorSum=0 --初始化为 0，即无错误
5. /*--转账：张三的账户少 1000 元，李四的账户多 1000 元*/
6. UPDATE bank SET currentMoney=currentMoney-1000
7. WHERE customerName='张三'
8. SET @errorSum=@errorSum+@@error
9. UPDATE bank SET currentMoney=currentMoney+1000
10. WHERE customerName='李四'
11.SET @errorSum=@errorSum+@@error --累计是否有错误
12.IF @errorSum<>0 --如果有错误
13. BEGIN
14. print '交易失败，回滚事务'
15. ROLLBACK TRANSACTION
16. END
17.ELSE
18. BEGIN
19. print '交易成功，提交事务，写入硬盘，永久的保存'
```

```
20. COMMIT TRANSACTION
21. END
22.GO
23.print '查看转账事务后的余额'
24.SELECT * FROM bank
25.GO
```



The screenshot shows a SQL Server query result window with the following content:

```
结果
查看转账事务前的余额
customerName currentMoney
-----
张三          1000.00
李四          1.00

消息 547, 级别 16, 状态 0, 第 7 行
UPDATE 语句与 CHECK 约束"CK_currentMoney"冲突。
该冲突发生于数据库"stuDB", 表"dbo.bank", column 'currentMoney'。
语句已终止。
查看转账事务过程中的余额
customerName currentMoney
-----
张三          1000.00
李四          1001.00

交易失败, 回滚事务
查看转账事务后的余额
customerName currentMoney
-----
张三          1000.00
李四          1.00
```

Java调用数据库事务方法在 [ava 程序员从笨鸟到菜鸟之（七）——java 数据库操作](#)

已经提到过了。在此就不在陈述了

二：索引

首先看一下什么事索引（以 **sqlserver** 为例）：

SQL Server 中的数据也是按页（ 4KB ）存放

索引：是 SQL Server 编排数据的内部方法。它为 SQL Server 提供一种方法来编排查询数据

索引页：数据库中存储索引的数据页；索引页类似于汉语字（词）典中按拼音或笔画排序的目录页

索引的作用：通过使用索引，可以大大提高数据库的检索速度，改善数据库性能

然后看一下索引的类型：

- 1.唯一索引：唯一索引不允许两行具有相同的索引值
- 2.主键索引：为表定义一个主键将自动创建主键索引，主键索引是唯一索引的特殊类型。主键索引要求主键中的每个值是唯一的，并且不能为空
- 3.聚集索引(Clustered)：表中各行的物理顺序与键值的逻辑（索引）顺序相同，每个表只能有一个
- 4.非聚集索引(Non-clustered)：非聚集索引指定表的逻辑顺序。数据存储在一个位置，索引存储在另一个位置，索引中包含指向数据存储位置的指针。可以有多个，小于 249 个

示例：利用企业管理器创建索引



使用 T-SQL 语句创建索引的语法：

```
CREATE [UNIQUE] [CLUSTERED|NONCLUSTERED]
INDEX index_name
ON table_name (column_name...)
[WITH FILLFACTOR=x]
```

注：UNIQUE 表示唯一索引，可选

CLUSTERED、NONCLUSTERED 表示聚集索引还是非聚集索引，可选

FILLFACTOR 表示填充因子，指定一个 0 到 100 之间的值，该值指示索引页填满的空间所占的百分比

创建索引示例：

[sql] [view plaincopyprint?](#)

```
1. USE stuDBGOIF EXISTS (SELECT name FROM sysindexes WHERE name = 'IX_writtenExam') DROP INDEX stuMarks.IX_writtenExam /*--笔试题创建非聚集索引：填充因子为 30%
--*/CREATE NONCLUSTERED INDEX IX_writtenExam ON stuMarks(writtenExam) WITH FILLFACTOR= 30GO/*-----指定按索引 IX_writtenExam 查询
----*/SELECT * FROM stuMarks with (INDEX=IX_writtenExam) WHERE writtenExam BETWEEN 60 AND 90
```

索引的优缺点：

优点：1.加快访问速度 2.加强行的唯一性

缺点：1.带索引的表在数据库中需要更多的存储空间

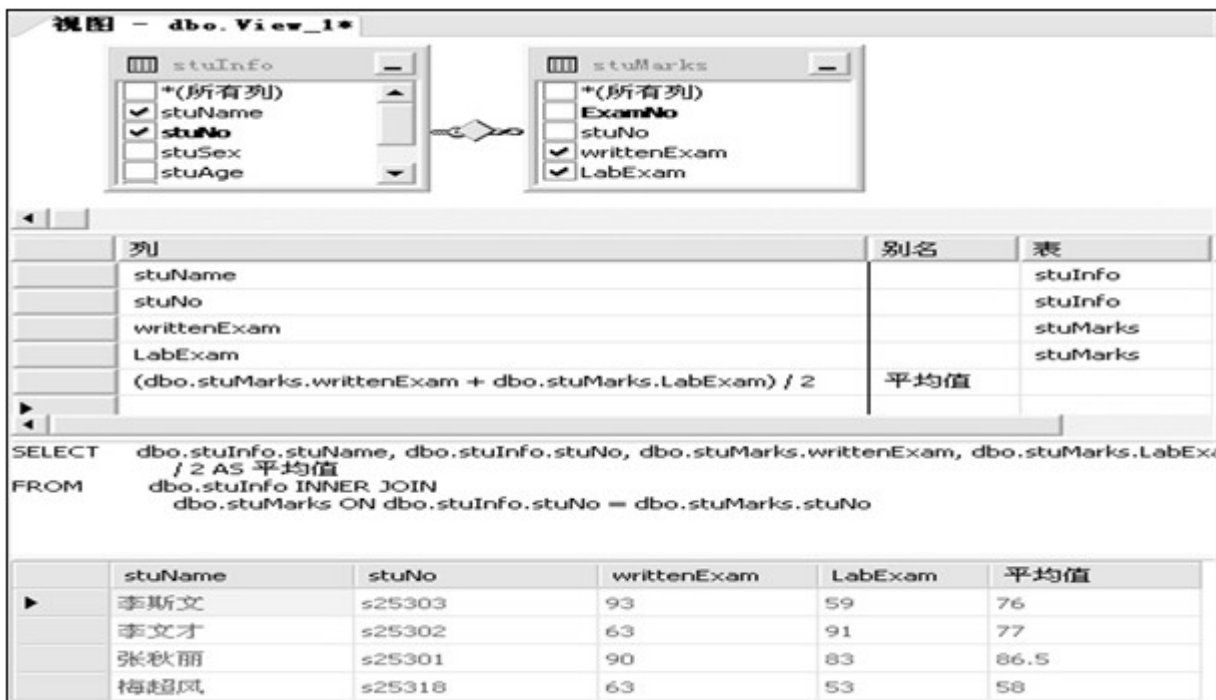
2.操纵数据的命令需要更长的处理时间，因为它们需要对索引进行更新

三：视图

首先还是先看一下什么事视图：

视图是一张虚拟表，它表示一张表的部分数据或多张表的综合数据，其结构和数据是建立在对表的查询基础上。视图中并不存放数据，而是存放在视图所引用的原始表（基表）中同一张原始表，根据不同用户的不同需求，可以创建不同的视图

使用企业管理器创建视图：



使用 T-SQL 语句创建视图的语法：

```

CREATE VIEW view_name
AS
<select 语句>
    
```

示例：创建方便教员查看成绩的视图

[sql] view plaincopyprint?

1. `IF EXISTS (SELECT * FROM sysobjects WHERE name = 'view_stuInfo_stuMarks')`
2. `DROP VIEW view_stuInfo_stuMarks`
3. `GO`
4. `CREATE VIEW view_stuInfo_stuMarks`

```

5. CREATE VIEW view_stuInfo_stuMarks
6. AS
7. SELECT 姓名=stuName,学号=stuInfo.stuNo,
8.    笔试成绩 =writtenExam, 机试成绩=labExam,
9.    平均分=(writtenExam+labExam)/2
10. FROM stuInfo LEFT JOIN stuMarks
11.    ON stuInfo.stuNo=stuMarks.stuNo
12.GO
13.SELECT * FROM view_stuInfo_stuMarks</SPAN>

```

四：存储过程：

首先还是来看一下什么事存储过程：

存储过程（Stored Procedure）是在大型数据库系统中，一组为了完成特定功能的 SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。

存储过程（procedure）类似于 Java 语言中的方法

用来执行管理任务或应用复杂的业务规则

存储过程可以带参数，也可以返回结果

存储过程的有点：

执行速度更快

允许模块化程序设计

提高系统安全性

减少网络流通量

存储过程分类：

1.系统存储过程

由系统定义，存放在 master 数据库中，类似 Java 语言类库中的方法。

系统存储过程的名称都以“sp_”开头或“xp_”开头

2.用户自定义存储过程

由用户在自己的数据库中创建的存储过程，类似 Java 语言中用户自定义的方法

常用的系统存储过程

系统存储过程	说明
sp_databases	列出服务器上的所有数据库
sp_helpdb	报告有关指定数据库或所有数据库的信息
sp_renamedb	更改数据库的名称
sp_tables	返回当前环境下可查询的对象的列表
sp_columns	返回某个表列的信息
sp_help	查看某个表的所有信息
sp_helpconstraint	查看某个表的约束
sp_helpindex	查看某个表的索引
sp_stored_procedures	列出当前环境中的所有存储过程
sp_password	添加或修改登录帐户的密码
sp_helptext	显示默认值、未加密的存储过程、用户定义的存储过程、触发器或视图的实际文本

定义存储过程的语法

CREATE PROC[EDURE] 存储过程名

 @参数 1 数据类型 = 默认值 OUTPUT,

 ,

 @参数 n 数据类型 = 默认值 OUTPUT

AS

SQL 语句

GO

和 Java 语言的方法一样，参数可选

参数分为输入参数、输出参数

输入参数允许有默认值

先给出一个不带输入参数的存储过程的例子：

[sql] [view plaincopyprint?](#)

```
1. CREATE PROCEDURE proc_stu //proc_stu 为存储过程的名称
2. AS DECLARE @writtenAvg float,@labAvg float //笔试平均分和机试平均分变量
   SELECT @writtenAvg=AVG(writtenExam), @labAvg=AVG(labExam)
   FROM stuMarks print '笔试平均分: '+convert(varchar(5),@writtenAvg) print '
   机试平均分:
   '+convert(varchar(5),@labAvg) IF (@writtenAvg>70 AND @labAvg>70) print '
   本班考试成绩: 优秀' ELSE print '本班考试成绩: 较差
   ' print '-----' print '      参加本次考试没有通过的学员:
   ' SELECT stuName,stuInfo.stuNo,writtenExam,labExam FROM stuInfo INNER JOIN stuMarks ON
   stuInfo.stuNo=stuMarks.stuNo WHERE writtenExam<60 OR labExam<60 GO
```

执行存储过程的语法：

调用的语法

EXEC 过程名 [参数]

存储过程的参数分两种：1.输入参数 2.输出参数

输入参数：用于向存储过程传入值，类似 Java 带参方法

输出参数：用于在调用存储过程后，返回结果

带输入参数的存储过程：

修改上例子：由于每次考试的难易程度不一样，每次笔试和机试的及格线可能随时变化（不再是 60 分），这导致考试的评判结果也相应变化

调用上面的存储过程：

[sql] [view plaincopyprint?](#)


```

1. CREATE PROCEDURE proc_stu @writtenPass int, //输入参数：笔试及格
   线 @labPass int //输入参数：机试及格
   线 AS print '-----' print ' 参加本次考试
   没有通过的学员：
   ' SELECT stuName,stuInfo.stuNo,writtenExam, labExam FROM stuInfo
   INNER JOIN stuMarks ON //查询没有通过考试的学
   员 stuInfo.stuNo=stuMarks.stuNo WHERE writtenExam<@writt
   enPass OR labExam<@labPass GO

```

EXEC proc_stu 60,55

--或这样调用：

EXEC proc_stu @labPass=55,@writtenPass=60

扩展：设置输入的默认值：

--或这样调用：

```

EXEC proc_stu
  @labPass=55,
  @writtenPass=60
CREATE PROCEDURE proc_stu
  @writtenPass int=60,
  @labPass int=60
AS

```

。 。 。 。 。 。

调用带参数默认值的存储过程

EXEC proc_stu --都采用默认值

EXEC proc_stu 64 --机试采用默认值

EXEC proc_stu 60,55 --都不采用默认值

--错误的调用方式：希望笔试采用默认值，机试及格线 55 分

EXEC proc_stu ,55

--正确的调用方式:

EXEC proc_stu @labPass=55

如果希望调用存储过程后, 返回一个或多个值, 这时就需要使用输出

(OUTPUT) 参数了

[html] view plaincopyprint?

```
1. CREATE PROCEDURE proc_stu @notpassSum int OUTPUT, //输出 (返回) 参
   数: 表示没有通过的人
   数 @writtenPass int= , @labPass int= AS ..... SELECT stuName,stu
   Info.stuNo,writtenExam, labExam FROM stuInfo INNER JOIN stuMarks
   ON stuInfo.stuNo= .stuNo WHERE writtenExam<@writtenPas
   s OR labExam<@labPass SELECT @notpassSum= (stuNo) /
   /统计并返回没有通过考试的学员人
   数 FROM stuMarks WHERE writtenExam<@writtenPass OR labExam
   <@labPass GO
```

调用带输出参数的存储过程

[sql] view plaincopyprint?

```
1. /*---调用存储过程
   ----*/DECLARE @sum int EXEC proc_stu @sum OUTPUT ,64 //调用时必须带
   OUTPUT 关键字 , 返回结果将存放在变量@sum
   中 print '-----'IF @sum>=3 //后续语句引用
   返回结果 print '未通过人数: '+convert(varchar(5),@sum)+'人, 超过 60%,及
   格分数线还应下调'ELSE print '未通过人数: '+convert(varchar(5),@sum)+'
   人, 已控制在 60%以下, 及格分数线适中'GO
```

（十）枚举，泛型详解

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

一：首先从枚举开始说起

枚举类型是 JDK5.0 的新特征。Sun 引进了一个全新的关键字 `enum` 来定义一个枚举类。下面就是一个典型枚举类型的定义：

```
public enum Color{  
    RED, BLUE, BLACK, YELLOW, GREEN  
}
```

显然，`enum` 很像特殊的 `class`，实际上 `enum` 声明定义的类型就是一个类。而这些类都是类库中 `Enum` 类的子类（`java.lang.Enum`）。它们继承了这个 `Enum` 中的许多有用的方法。我们对代码编译之后发现，编译器将 `enum` 类型单独编译成了一个字节码文件：`Color.class`。

Color 字节码代码

```
final enum hr.test.Color {  
    // 所有的枚举值都是类静态常量  
  
    public static final enum hr.test.Color RED;  
    public static final enum hr.test.Color BLUE;  
    public static final enum hr.test.Color BLACK;  
    public static final enum hr.test.Color YELLOW;  
    public static final enum hr.test.Color GREEN;  
    private static final synthetic hr.test.Color []    ENUM$VALUES;
```

```
}
```

下面我们就详细介绍 `enum` 定义的枚举类的特征及其用法。（后面均用 `Color` 举例）

1、`Color` 枚举类就是 `class`，而且是一个不可以被继承的 `final` 类。其枚举值（`RED`，`BLUE`...）都是 `Color` 类型的类静态常量， 我们可以通过下面的方式来得到 `Color` 枚举类的一个实例：

```
Color c=Color.RED;
```

注意：这些枚举值都是 `public static final` 的，也就是我们经常所定义的常量方式，因此枚举类中的枚举值最好全部大写。

2、即然枚举类是 `class`，当然在枚举类型中有构造器，方法和数据域。但是，枚举类的构造器有很大的不同：

（1） 构造器只是在构造枚举值的时候被调用。

Java 代码

```
enum Color{
    RED (255, 0, 0) , BLUE (0, 0, 255) , BLACK (0, 0, 0) , YELLOW
    (255, 255, 0) , GREEN (0, 255, 0) ;

    //构造枚举值，比如 RED (255, 0, 0)

    private Color (int rv, int gv, int bv) {
        this.redValue=rv;
        this.greenValue=gv;
        this.blueValue=bv;
    }

    public String toString () { //覆盖了父类 Enum 的 toString ()

        return super.toString()+"(" +redValue+" , "+greenValue+" , "+blueValue+" )";
    }
}
```

```

}
private int redValue; //自定义数据域，private 为了封装。
private int greenValue;
private int blueValue;
}

```

(2) 构造器只能私有 `private`，绝对不允许有 `public` 构造器。这样可以保证外部代码无法新构造枚举类的实例。这也是完全符合情理的，因为我们知道枚举值是 `public static final` 的常量而已。但枚举类的方法和数据域可以允许外部访问。

Java 代码

```

public static void main (String args [])
{
// Color colors=new Color (100, 200, 300) ;//wrong
Color color=Color.RED;
System.out.println (color) ;// 调用了 toString () 方法
}

```

3、所有枚举类都继承了 `Enum` 的方法，下面我们详细介绍这些方法。

(1) `ordinal ()` 方法： 返回枚举值在枚举类种的顺序。这个顺序根据枚举值声明的顺序而定。

```
Color.RED.ordinal () ;//返回结果： 0
```

```
Color.BLUE.ordinal () ;//返回结果： 1
```

(2) `compareTo ()` 方法： `Enum` 实现了 `java.lang.Comparable` 接口，因此可以比较象与指定对象的顺序。`Enum` 中的 `compareTo` 返回的是两个枚举值的顺序之差。当然，前提是两个枚举值必须属于同一个枚举类，否则会抛出 `ClassCastException ()` 异常。（具体可见源代码）

`Color.RED.compareTo (Color.BLUE) ;//返回结果 -1`

(3) `values ()` 方法: 静态方法, 返回一个包含全部枚举值的数组。

```
Color [] colors=Color.values () ;
```

```
for (Color c:colors) {
```

```
System.out.print (c+" , ") ;
```

```
//返回结果: RED, BLUE, BLACK YELLOW, GREEN,
```

(4) `toString ()` 方法: 返回枚举常量的名称。

```
Color c=Color.RED;
```

```
System.out.println (c) ;//返回结果: RED
```

(5) `valueOf ()` 方法: 这个方法和 `toString` 方法是相对应的, 返回带指定名称的指定枚举类型的枚举常量。

```
Color.valueOf ("BLUE") ;//返回结果: Color.BLUE
```

(6) `equals ()` 方法: 比较两个枚举类对象的引用。

Java 代码

//JDK 源代码:

```
public final boolean equals (Object other) {
```

```
return this==other;
```

```
}
```

4、枚举类可以在 `switch` 语句中使用。

Java 代码

```
Color color=Color.RED;
```

```
switch (color) {
```

```
case RED: System.out.println ("it's red") ;break;
```

```
case BLUE: System.out.println ("it's blue") ;break;
```

```
case BLACK:    System.out.println (“it’s blue”);break;
}
```

二：然后看泛型

泛型（Generic type 或者 generics）是对 Java 语言的类型系统的一种扩展，以支持创建可以按类型进行参数化的类。可以把类型参数看作是使用参数化类型时指定的类型的一个占位符，就像方法的形式参数是运行时传递的值的占位符一样。

1.泛型的好处：

1) 类型安全。泛型的主要目标是提高 Java 程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在一个高得多的程度上验证类型假设。没有泛型，这些假设就只存在于程序员的头脑中（或者如果幸运的话，还存在于代码注释中）。

2) ·消除强制类型转换。泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。 尽管减少强制类型转换可以降低使用泛型类的代码的罗嗦程度，但是声明泛型变量会带来相应的罗嗦

3) ·潜在的性能收益。泛型为较大的优化带来可能。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，程序员会指定这些强制类型转换）插入生成的字节码中。但是更多类型信息可用于编译器这一事实，为未来版本的 JVM 的优化带来可能。

2.类型参数：

在定义泛型类或声明泛型类的变量时，使用尖括号来指定形式类型参数。形

式类型参数与实际类型参数之间的关系类似于形式方法参数与实际方法参数之间的关系，只是类型参数表示类型，而不是表示值。

泛型类中的类型参数几乎可以用于任何可以使用类名的地方。例如，下面是 `java.util.Map` 接口的定义的摘录：

```
public interface Map<K, V> {  
    public void put(K key, V value);  
    public V get(K key);  
}
```

3. 泛型不是协变的

关于泛型的混淆，一个常见的来源就是假设它们像数组一样是协变的。其实它们不是协变的。`List<Object>` 不是 `List<String>` 的父类型。

如果 `A` 扩展 `B`，那么 `A` 的数组也是 `B` 的数组，并且完全可以在需要 `B[]` 的地方使用 `A[]`：

```
Integer[] intArray = new Integer[10];  
Number[] numberArray = intArray;
```

上面的代码是有效的，因为一个 `Integer` 是一个 `Number`，因而一个 `Integer` 数组是一个 `Number` 数组。但是对于泛型来说则不然。下面的代码是无效的

```
List<Integer> intList = new ArrayList<Integer>();  
List<Number> numberList = intList; // invalid
```

4. 泛型中的类型通配符

假设您具有该方法：

```
void printList(List l) {  
    for (Object o : l)  
        System.out.println(o);  
}
```

上面的代码在 `JDK 5.0` 上编译通过，但是如果试图用 `List<Integer>` 调用它，

则会得到警告。出现警告是因为，您将泛型（`List<Integer>`）传递给一个只

承诺将它当作 List（所谓的原始类型）的方法，这将破坏使用泛型的类型安全。

如果试图编写像下面这样的方法，那么将会怎么样？

```
void printList(List<Object> l) {  
    for (Object o : l)  
        System.out.println(o);  
}
```

它仍然不会通过编译，因为一个 List<Integer>不是一个 List<Object>（正如前一屏泛型不是协变的 中所学的）。这才真正烦人——现在您的泛型版本还没有普通的非泛型版本有用！ 解决方案是使用类型通配符：

```
void printList(List<?> l) {  
    for (Object o : l)  
        System.out.println(o);  
}
```

上面代码中的问号是一个类型通配符。它读作“问号”。List<?>是任何泛型 List 的父类型，所以您完全可以将 List<Object>、List<Integer> 或 List<List<List<Flutzpah>>>传递给 printList()。

5.泛型方法

（在类型参数 一节中）您已经看到，通过在类的定义中添加一个形式类型参数列表，可以将类泛型化。方法也可以被泛型化，不管它们定义在其中的类是不是泛型化的。

泛型类在多个方法签名间实施类型约束。在 List<V>中，类型参数 V 出现在 get()、add()、contains()等方法的签名中。 当创建一个 Map<K, V>类型的变量时，您就在方法之间宣称一个类型约束。您传递给 add()的值将与 get()返回

的值的类型相同。

类似地，之所以声明泛型方法，一般是因为您想要在该方法的多个参数之间宣称一个类型约束。例如，下面代码中的 `ifThenElse()` 方法，根据它的第一个参数的布尔值，它将返回第二个或第三个参数：

```
public <T> T ifThenElse(boolean b, T first, T second) {  
    return b ? first : second;  
}
```

为什么您选择使用泛型方法，而不是将类型 `T` 添加到类定义呢？（至少）有两种情况应该这样做：

- * 当泛型方法是静态的时，这种情况下不能使用类类型参数。
- * 当 `T` 上的类型约束对于方法真正是局部的时，这意味着没有在不同类的另一个方法签名中使用相同类型 `T` 的约束。通过使得泛型方法的类型参数对于方法是局部的，可以简化封闭类型的签名。

有限制类型

在前一屏泛型方法的例子中，类型参数 `V` 是无约束的或无限制的类型。有时在还没有完全指定类型参数时，需要对类型参数指定附加的约束。

考虑例子 `Matrix` 类，它使用类型参数 `V`，该参数由 `Number` 类来限制：

```
public class Matrix<V extends Number> { ... }
```

编译器允许您创建 `Matrix<Integer>` 或 `Matrix<Float>` 类型的变量，但是如果您试图定义 `Matrix<String>` 类型的变量，则会出现错误。类型参数 `V` 被判断为由 `Number` 限制。在没有类型限制时，假设类型参数由 `Object` 限制。这就是为什么前一屏泛型方法中的例子，允许 `List.get()` 在 `List<?>` 上调用时返回 `Object`，即使编译器不知道类型参数 `V` 的类型。

(十一) 多线程讲解

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

多线程是 java 应用程序的一个特点，掌握 java 的多线程也是作为一 java 程序员必备的知识。多线程指的是在单个程序中可以同时运行多个同的线程执行不同的任务.线程是程序内的顺序控制流，只能使用分配给序的资源 and 环境。还记得刚开始学习的时候总是和进程分不清，总是对这两个名词所迷惑。下面就首先对这两个名词区分来作为本篇博客的开始：

一、线程与进程的区别

多个进程的 internal 数据和状态都是完全独立的,而多线程是共享一块内存空间 and 一组系统资源,有可能互相影响. •线程本身的数据通常只有寄存器数据，以及一个程序执行时使用的堆栈，所以线程的切换比进程切换的负担要小。

多线程编程的目的，就是"最大限度地利用 CPU 资源"，当某一线程的处理不需要占用 CPU 而只和 I/O 等资源打交道时，让需要占用 CPU 资源的其它线程有机会获得 CPU 资源。从根本上说，这就是多线程编程的最终目的。

二、了解一下 java 在多线程中的基础知识

1.Java 中如果我们自己没有产生线程，那么系统就会给我们产生一个线程（主线程，main 方法就在主线程上运行），我们的程序都是由线程来执行的。

2. 进程：执行中的程序（程序是静态的概念，进程是动态的概念）。

3. 线程的实现有两种方式，第一种方式是继承 **Thread** 类，然后重写 **run** 方法；第二种是实现 **Runnable** 接口，然后实现其 **run** 方法。

4. 将我们希望线程执行的代码放到 **run** 方法中，然后通过 **start** 方法来启动线程，**start** 方法首先为线程的执行准备好系统资源，然后再去调用 **run** 方法。

当某个类继承了 **Thread** 类之后，该类就叫做一个线程类。

5. 一个进程至少要包含一个线程。

6. 对于单核 **CPU** 来说，某一时刻只能有一个线程在执行（微观串行），从宏观角度来看，多个线程在同时执行（宏观并行）。

7. 对于双核或双核以上的 **CPU** 来说，可以真正做到微观并行。

三、Thread 源码研究：

1) **Thread** 类也实现了 **Runnable** 接口，因此实现了 **Runnable** 接口中的 **run** 方法；

2) 当生成一个线程对象时，如果没有为其设定名字，那么线程对象的名字将使用如下形式：**Thread-number**, 该 **number** 将是自动增加的，并被所有的 **Thread** 对象所共享（因为它是 **static** 的成员变量）。

3) 当使用第一种方式来生成线程对象时，我们需要重写 **run** 方法，因为 **Thread** 类的 **run** 方法此时什么事情也不做。

4) 当使用第二种方式生成线程对象时，我们需要实现 **Runnable** 接口的 **run** 方法，然后使用 **new Thread (new MyThread ())**（假如 **MyThread** 已经实现了 **Runnable** 接口）来生成线程对象，这时的线程对象的 **run** 方法或调就会 **MyThread** 类的 **run** 方法，这样我们自己编写的 **run** 方法就执行了。

说明：

```
Public void run(){  
If(target!=null){  
Target.run();
```

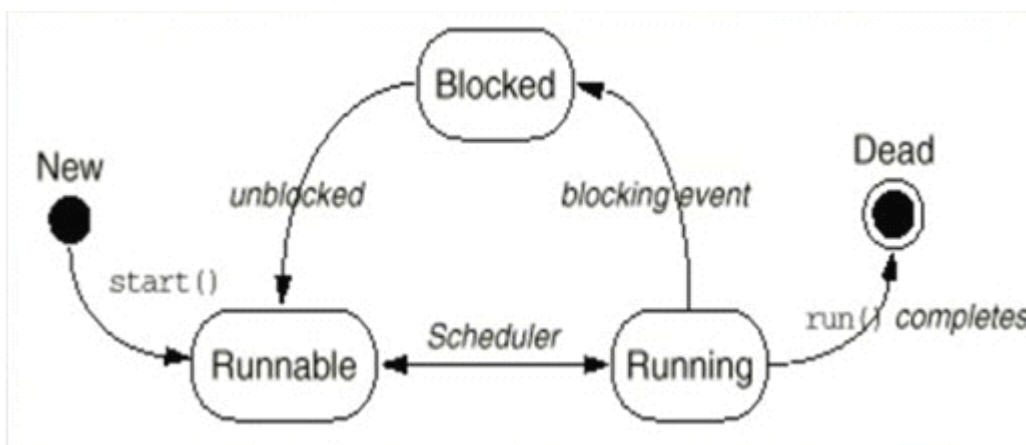
}}

当使用继承 **Thread** 生成线程对象时，**target** 为空，什么也不执行，当使用第二种方式生成时，执行 **target.run ()**，**target** 为 **runnable** 的实例对象，即为执行重写后的方法

总结：两种生成线程对象的区别：

- 1.两种方法均需执行线程的 **start** 方法为线程分配必须的系统资源、调度线程运行并执行线程的 **run** 方法。
- 2.在具体应用中，采用哪种方法来构造线程体要视情况而定。通常，当一个线程已继承了另一个类时，就应该用第二种方法来构造，即实现 **Runnable** 接口。

四：线程的生命周期：



由上图可以看出，一个线程由出生到死亡分为五个阶段：

1) .创建状态

- 当用 **new** 操作符创建一个新的线程对象时，该线程处于创建状态。
- 处于创建状态的线程只是一个空的线程对象，系统不为它分配资源

2) .可运行状态

- 执行线程的 **start()**方法将为线程分配必须的系统资源，安排其运行，并调用线程体—**run()**方法，这样就使得该线程处于可运行(**Runnable**)状态。
- 这一状态并不是运行中状态 (**Running**)，因为线程也许实际上并未真正运行。

3) .不可运行状态

.当发生下列事件时，处于运行状态的线程会转入到不可运行状态。

调用了 **sleep ()** 方法；

- 线程调用 **wait** 方法等待特定条件的满足
- 线程输入/输出阻塞

4) 返回可运行状态：

- 处于睡眠状态的线程在指定的时间过去后
- 如果线程在等待某一条件，另一个对象必须通过 **notify()**或 **notifyAll()**方法通知等待线程条件的改变
- 如果线程是因为输入/输出阻塞，等待输入/输出完成

5) .消亡状态

当线程的 **run** 方法执行结束后，该线程自然消亡。

注意：

- 1.停止线程的方式：不能使用 **Thread** 类的 **stop** 方法来终止线程的执行。一般要设定一个变量，在 **run** 方法中是一个循环，循环每次检查该变量，如果满足条件则继续执行，否则跳出循环，线程结束。
- 2.不能依靠线程的优先级来决定线程的执行顺序。

五：多线程并发

多线程并发是线程同步中比较常见的现象，java 多线程为了避免多线程并发解决多线程共享数据同步问题提供了 **synchronized 关键字**

synchronized 关键字：当 **synchronized** 关键字修饰一个方法的时候，该方法叫做同步方法。

1. **Java** 中的每个对象都有一个锁（**lock**）或者叫做监视器（**monitor**），当访问某个对象的 **synchronized** 方法时，表示将该对象上锁，此时其他任何线程都无法再去访问该 **synchronized** 方法了，直到之前的那个线程执行方法完毕后（或者是抛出了异常），那么将该对象的锁释放掉，其他线程才有可能再去访问该 **synchronized** 方法。

2. 如果一个对象有多个 **synchronized** 方法，某一时刻某个线程已经进入到了某个 **synchronized** 方法，那么在该方法没有执行完毕前，其他线程是无法访问该对象的任何 **synchronized** 方法的。

3. 如果某个 **synchronized** 方法是 **static** 的，那么当线程访问该方法时，它锁的并不是 **synchronized** 方法所在的对象，而是 **synchronized** 方法所在的对象所对应的 **Class** 对象，因为 **Java** 中无论一个类有多少个对象，这些对象会对应唯一一个 **Class** 对象，因此当线程分别访问同一个类的两个对象的两个 **static**，**synchronized** 方法时，他们的执行顺序也是顺序的，也就是说一个线程先去执行方法，执行完毕后另一个线程才开始执行。

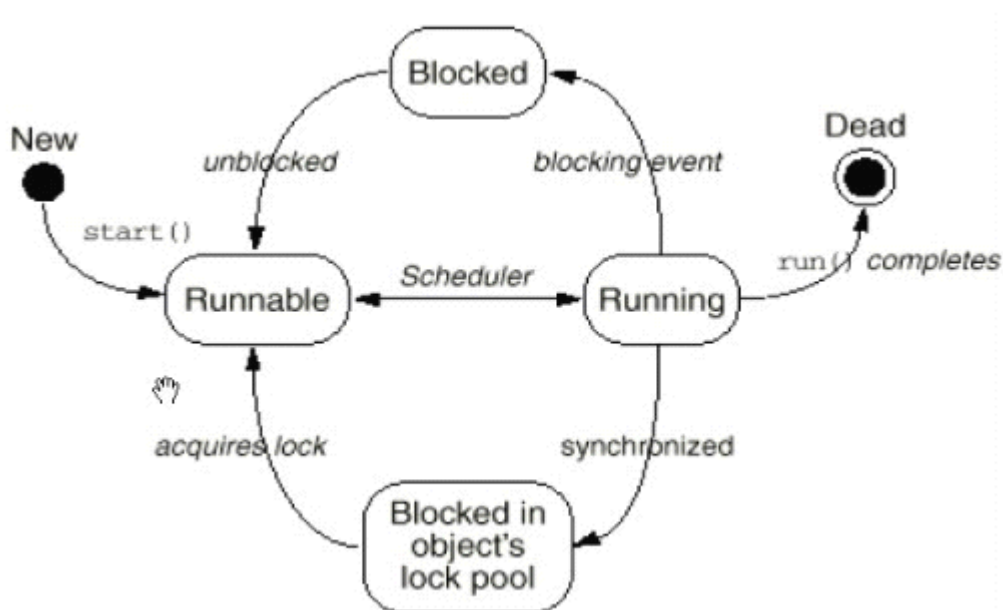
4. **synchronized** 块，写法：

```
synchronized(object)
{
}
```

表示线程在执行的时候会对 **object** 对象上锁。

5.synchronized 方法是一种粗粒度的并发控制，某一时刻，只能有一个线程执行该 synchronized 方法;synchronized 块则是一种细粒度的并发控制，只会将块中的代码同步，位于方法内、synchronized 块之外的代码是可以被多个线程同时访问到的。

同步的线程状态图：



六：wait 与 notify

1.wait 与 notify 方法都是定义在 Object 类中，而且是 final 的，因此会被所有的 Java 类所继承并且无法重写。这两个方法要求在调用时线程应该已经获得了对象的锁，因此对这两个方法的调用需要放在 synchronized 方法或块当中。当线程执行了 wait 方法时，它会释放掉对象的锁。

2. 另一个会导致线程暂停的方法就是 Thread 类的 sleep 方法，它会导致线程睡眠指定的毫秒数，但线程在睡眠的过程中是不会释放掉对象的锁的。

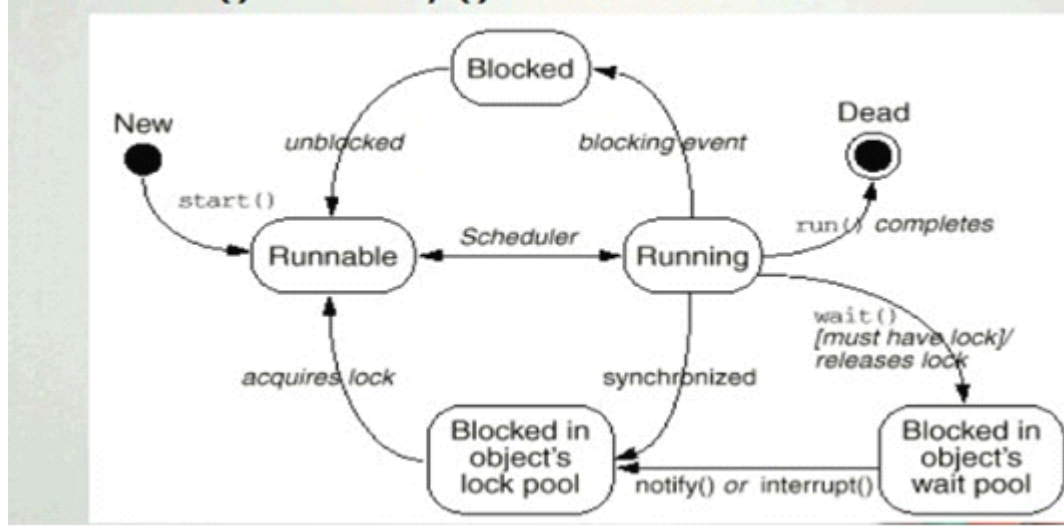
3.notify():唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。

直到当前线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。此方法只应由作为此对象监视器的所有者的线程来调用。通过以下三种方法之一，线程可以成为此对象监视器的所有者：

- o 通过执行此对象的同步实例方法。
- o 通过执行在此对象上进行同步的 `synchronized` 语句的正文。
- o 对于 `Class` 类型的对象，可以通过执行该类的同步静态方法。

一次只能有一个线程拥有对象的监视器。

具有`wait()`和`notify()`的线程状态图：



关于成员变量与局部变量：如果一个变量是成员变量，那么多个线程对同一个对象的成员变量进行操作时，他们对该成员变量是彼此影响的（也就是说一个线程对成员变量的改变会影响到另一个线程）。如果一个变量是局部变量，那么每个线程都会有一个该局部变量的拷贝，一个线程对该局部变量的改变不会影响到其他的线程。

七：死锁的问题：

定义：线程 1 锁住了对象 A 的监视器，等待对象 B 的监视器，线程 2 锁住了对象 B 的监视器，等待对象 A 的监视器，就造成了死锁。

导致死锁的根源在于不适当地运用“synchronized”关键词来管理线程对特定对象的访问。“synchronized”关键词的作用是，确保在某个时刻只有一个线程被允许执行特定的代码块，因此，被允许执行的线程首先必须拥有对变量或对象的排他性访问权。当线程访问对象时，线程会给对象加锁

Java 中每个对象都有一把锁与之对应。但 Java 不提供单独的 lock 和 unlock 操作。下面笔者分析死锁的两个过程“上锁”和“锁死”。

(1) 上锁

许多线程在执行中必须考虑与其他线程之间共享数据或协调执行状态，就需要同步机制。因此大多数应用程序要求线程互相通信来同步它们的动作，在 Java 程序中最简单实现同步的方法就是上锁。在 Java 编程中，所有的对象都有锁。线程可以使用 **synchronized** 关键字来获得锁。在任一时刻对于给定的类的实例，方法或同步的代码块只能被一个线程执行。这是因为代码在执行之前要求获得对象的锁。

为了防止同时访问共享资源，线程在使用资源的前后可以给该资源上锁和开锁。给共享变量上锁就使得 Java 线程能够快速方便地通信和同步。某个线程若给一个对象上了锁，就可以知道没有其他线程能够访问该对象。即使在抢占式模型中，其他线程也不能够访问此对象，直到上锁的线程被唤醒、完成工作并开锁。那些试图访问一个上锁对象的线程通常会进入睡眠状态，直到上锁的线程开锁。一旦锁被打开，这些睡眠进程就会被唤醒并移到准备就绪队列中。

(2)锁死

如果程序中有几个竞争资源的并发线程,那么保证均衡是很重要的。系统均衡是指每个线程在执行过程中都能充分访问有限的资源，系统中没有饿死和死锁的线程。当多个并发的线程分别试图同时占有两个锁时，会出现加锁冲突的情形。如果一个线程占有了另一个线程必需的锁，互相等待时被阻塞就有可能出现死锁。

在编写多线程代码时，笔者认为死锁是最难处理的问题之一。因为死锁可能在最意想不到的地方发生，所以查找和修正它既费时又费力。例如，常见的例子如下面这段程序。**print?**

```
1 public int sumArrays(int[] a1, int[] a2){
2   int value = 0;
3   int size = a1.length;
4   if (size == a2.length) {
5     synchronized(a1) { //1
6       synchronized(a2) { //2
7         for (int i=0; i<size; i++)
8           value += a1[i] + a2[i];
9       }
10    }
11  } return value;
12 }
```

这段代码在求和操作中访问两个数组对象之前锁定了这两个数组对象。它形式简短，编写也适合所要执行的任务；但不幸的是，它有一个潜在的问题。这个问题就是它埋下了死锁的种子。

ThreadLocal 类（这个类本人没用过，占时不太懂）

首先，ThreadLocal 不是用来解决共享对象的多线程访问问题的，一般情况下，通过 ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其

他线程是不需要访问的，也访问不到的。各个线程中访问的是不同的对象。

另外，说 `ThreadLocal` 使得各线程能够保持各自独立的一个对象，并不是通过 `ThreadLocal.set()` 来实现的，而是通过每个线程中的 `new` 对象 的操作来创建的对象，每个线程创建一个，不是什么对象的拷贝或副本。通过 `ThreadLocal.set()` 将这个新创建的对象引用保存到各线程的自己的一个 `map` 中，每个线程都有这样一个 `map`，执行 `ThreadLocal.get()` 时，各线程从自己的 `map` 中取出放进去的对象，因此取出来的是各自自己线程中的对象，`ThreadLocal` 实例是作为 `map` 的 `key` 来使用的。

如果 `ThreadLocal.set()` 进去的东西本来就是多个线程共享的同一个对象，那么多个线程的 `ThreadLocal.get()` 取得的还是这个共享对象本身，还是有并发访问问题。

(十二) java 异常处理机制

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

异常处理是程序设计中一个非常重要的方面，也是程序设计的一大难点，从 C 开始，你也许已经知道如何用 `if...else...` 来控制异常了，也许是自发的，然而这种控制异常痛苦，同一个异常或者错误如果多个地方出现，那么你每个地方都要做相同处理，感觉相当的麻烦！

Java 语言在设计当初就考虑到这些问题，提出异常处理的框架的方案，所有的异常都可以用一个类型来表示，不同类型的异常对应不同的子类异常(这里的异常包括错误概念)，定义异常处理的规范，在 1.4 版本以后增加了异常链机制，从而便于跟踪异常!这是 **Java** 语言设计者的高明之处，也是 Java 语言中的一个难点，下面是我对 Java 异常知识的一个总结，也算是资源回收一下。

一、Java 异常的基础知识

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。比如说，你的代码少了一个分号，那么运行出来结果是提示是错误 `java.lang.Error`;如果你用 `System.out.println(11/0)`，那么你是因为你用 0 做了除数，会抛出 `java.lang.ArithmeticException` 的异常。

有些异常需要做处理，有些则不需要捕获处理，后面会详细讲到。

天有不测风云，人有旦夕祸福，**Java** 的程序代码也如此。在编程过程中，首先应当尽可能去避免错误和异常发生，对于不可避免、不可预测的情况则在考虑异常发生时如何处理。

Java 中的异常用对象来表示。**Java** 对异常的处理是按异常分类处理的，不同异常有不同的分类，每种异常都对应一个类型(class)，每个异常都对应一个异常(类的)对象。

异常类从哪里来?有两个来源，一是 **Java** 语言本身定义的一些基本异常类型，二是用户通过继承 **Exception** 类或者其子类自己定义的异常。**Exception** 类及其子类是 **Throwable** 的一种形式，它指出了合理的应用程序想要捕获的条件。

异常的对象从哪里来呢?有两个来源，一是 **Java** 运行时环境自动抛出系统生成的异常，而不管你是否愿意捕获和处理，它总要被抛出!比如除数为 0 的异常。二是程序员自己抛出的异常，这个异常可以是程序员自己定义的，也可以是 **Java** 语言中定义的，用 **throw** 关键字抛出异常，这种异常常用来向调用者汇报异常的一些信息。

异常是针对方法来说的，抛出、声明抛出、捕获和处理异常都是在方法中进行的。

Java 异常处理通过 5 个关键字 **try**、**catch**、**throw**、**throws**、**finally** 进行管理。基本过程是用 **try** 语句块包住要监视的语句，如果在 **try** 语句块内出现异常，则异常会被抛出，你的代码在 **catch** 语句块中可以捕获到这个异常并做处理;还有以部分系统生成的异常在 **Java** 运行时自动抛出。你也可以通过

throws 关键字在方法上声明该方法要抛出异常，然后在方法内部通过 **throw** 抛出异常对象。**finally** 语句块会在方法执行 **return** 之前执行，一般结构如下：

```
try{  
    程序代码  
  
}catch(异常类型 1 异常的变量名 1){  
  
    程序代码  
  
}catch(异常类型 2 异常的变量名 2){  
  
    程序代码  
  
}finally{  
    程序代码  
  
}
```

catch 语句可以有多个，用来匹配多个异常，匹配上多个中一个后，执行 **catch** 语句块时候仅仅执行匹配上的异常。**catch** 的类型是 **Java** 语言中定义的或者程序员自己定义的，表示代码抛出异常的类型，异常的变量名表示抛出异常的对象的引用，如果 **catch** 捕获并匹配上了该异常，那么就可以直接用这个异常变量名，此时该异常变量名指向所匹配的异常，并且在 **catch** 代码块中可以直接引用。这一点非常非常的特殊和重要！

Java 异常处理的目的是提高程序的健壮性，你可以在 **catch** 和 **finally** 代码块中给程序一个修正机会，使得程序不因异常而终止或者流程发生以外的改变。同时，通过获取 **Java** 异常信息，也为程序的开发维护提供了方便，一般通过异常信息就很快就能找到出现异常的问题(代码)所在。

Java 异常处理是 **Java** 语言的一大特色，也是个难点，掌握异常处理可以让写的代码更健壮和易于维护。

二、Java 异常类类图

下面是这几个类的层次图：

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.Error
          java.lang.ThreadDeath
```

下面四个类的介绍来自 `java api` 文档。

1、Throwable

`Throwable` 类是 Java 语言中所有错误或异常的超类。只有当对象是此类(或其子类之一)的实例时，才能通过 `Java` 虚拟机或者 `Java throw` 语句抛出。类似地，只有此类或其子类之一才可以是 `catch` 子句中的参数类型。

两个子类的实例，`Error` 和 `Exception`，通常用于指示发生了异常情况。通常，这些实例是在异常情况的上下文中新近创建的，因此包含了相关的信息(比如堆栈跟踪数据)。

2、Exception

`Exception` 类及其子类是 `Throwable` 的一种形式，它指出了合理的应用程序想要捕获的条件，表示程序本身可以处理的异常。

3、Error

`Error` 是 `Throwable` 的子类，表示仅靠程序本身无法恢复的严重错误，用于指示合理的应用程序不应该试图捕获的严重问题。

在执行该方法期间，无需在方法中通过 `throws` 声明可能抛出但没有捕获的 `Error` 的任何子类，因为 `Java` 编译器不去检查它，也就是说，当程序中

可能出现这类异常时，即使没有用 `try...catch` 语句捕获它，也没有用 `throws` 字句声明抛出它，还是会编译通过。

4、`RuntimeException`

`RuntimeException` 是那些可能在 `Java` 虚拟机正常运行期间抛出的异常的超类。`Java` 编译器不去检查它，也就是说，当程序中可能出现这类异常时，即使没有用 `try...catch` 语句捕获它，也没有用 `throws` 字句声明抛出它，还是会编译通过，这种异常可以通过改进代码实现来避免。

5、`ThreadDeath`

调用 `Thread` 类中带有零参数的 `stop` 方法时，受害线程将抛出一个 `ThreadDeath` 实例。

仅当应用程序在被异步终止后必须清除时才应该捕获这个类的实例。如果 `ThreadDeath` 被一个方法捕获，那么将它重新抛出非常重要，因为这样才能让该线程真正终止。

如果没有捕获 `ThreadDeath`，则顶级错误处理程序不会输出消息。

虽然 `ThreadDeath` 类是“正常出现”的，但它只能是 `Error` 的子类而不是 `Exception` 的子类，因为许多应用程序捕获所有出现的 `Exception`，然后又将其放弃。

以上是对有关异常 `API` 的一个简单介绍，用法都很简单，关键在于理解异常处理的原理，具体用法参看 `Java API` 文档。

三、`Java` 异常处理机制

对于可能出现异常的代码，有两种处理办法：

第一、在方法中用 **try...catch** 语句捕获并处理异常，**catch** 语句可以有多个，用来匹配多个异常。例如：

```
public void p(int x){
    try{
        ...
    }catch(Exception e){
        ...
    }finally{
        ...
    }
}
```

第二、对于处理不了的异常或者要转型的异常，在方法的声明处通过 **throws** 语句抛出异常。例如：

```
public void test1() throws MyException{
    ...
    if(...){
        throw new MyException();
    }
}
```

如果每个方法都是简单的抛出异常，那么在方法调用方法的多层嵌套调用中，**Java** 虚拟机会从出现异常的方法代码块中往回找，直到找到处理该异常的代码块为止。然后将异常交给相应的 **catch** 语句处理。如果 **Java** 虚拟机追溯到方法调用栈最底部 **main()**方法时，如果仍然没有找到处理异常的代码块，将按照下面的步骤处理：

第一、调用异常的对象的 **printStackTrace()**方法，打印方法调用栈的异常信息。

第二、如果出现异常的线程为主线程，则整个程序运行终止;如果非主线程，则终止该线程，其他线程继续运行。

通过分析思考可以看出，越早处理异常消耗的资源和时间越小，产生影响的范围也越小。因此，不要把自己能处理的异常也抛给调用者。

还有一点，不可忽视：**finally** 语句在任何情况下都必须执行的代码，这样可以保证一些在任何情况下都必须执行代码的可靠性。比如，在数据库查询异常的时候，应该释放 JDBC 连接等等。**finally** 语句先于 **return** 语句执行，而不论其先后位置，也不管是否 **try** 块出现异常。**finally** 语句唯一不被执行的情况是方法执行了 **System.exit()** 方法。**System.exit()** 的作用是终止当前正在运行的 **Java** 虚拟机。**finally** 语句块中不能通过给变量赋新值来改变 **return** 的返回值，也建议不要在 **finally** 块中使用 **return** 语句，没有意义还容易导致错误。

最后还应该注意一下异常处理的语法规则：

第一、**try** 语句不能单独存在，可以和 **catch**、**finally** 组成 **try...catch...finally**、**try...catch**、**try...finally** 三种结构，**catch** 语句可以有一个或多个，**finally** 语句最多一个，**try**、**catch**、**finally** 这三个关键字均不能单独使用。

第二、**try**、**catch**、**finally** 三个代码块中变量的作用域分别独立而不能相互访问。如果要在三个块中都可以访问，则需要将变量定义到这些块的外面。

第三、多个 **catch** 块时候，**Java** 虚拟机会匹配其中一个异常类或其子类，就执行这个 **catch** 块，而不会再执行别的 **catch** 块。

第四、**throw** 语句后不允许有紧跟其他语句，因为这些没有机会执行。

第五、如果一个方法调用了另外一个声明抛出异常的方法，那么这个方法要么处理异常，要么声明抛出。

那怎么判断一个方法可能会出现异常呢?一般来说,方法声明的时候用了 **throws** 语句,方法中有 **throw** 语句,方法调用的方法声明有 **throws** 关键字。

throw 和 **throws** 关键字的区别

throw 用来抛出一个异常,在方法体内。语法格式为: **throw** 异常对象。

throws 用来声明方法可能会抛出什么异常,在方法名后,语法格式为:

throws 异常类型 1, 异常类型 2...异常类型 n。

四、如何定义和使用异常类

1、使用已有的异常类,假如为 `IOException`、`SQLException`。

```
try{
    程序代码
}catch(IOException ioe){
    程序代码
}catch(SQLException sqle){
    程序代码
}finally{
    程序代码
}
```

2、自定义异常类

创建 `Exception` 或者 `RuntimeException` 的子类即可得到一个自定义的异常类。例如:

```
public class MyException extends Exception{
    public MyException(){
    public MyException(String smg){
        super(smg);
    }
}
```

3、使用自定义的异常

用 **throws** 声明方法可能抛出自定义的异常，并用 **throw** 语句在适当的地方抛出自定义的异常。例如：

在某种条件抛出异常

```
public void test1() throws MyException{
    ...
    if(...){
        throw new MyException();
    }
}
```

将异常转型(也叫转译)，使得异常更易读易于理解

```
public void test2() throws MyException{
    ...
    try{
        ...
    }catch(SQLException e){
        ...
        throw new MyException();
    }
}
```

还有一个代码，很有意思：

```
public void test2() throws MyException{
    ...
    try {
        ...
    } catch (MyException e) {
        throw e;
    }
}
```

这段代码实际上捕获了异常，然后又和盘托出，没有一点意义，如果这样还有什么好处理的，不处理就行了，直接在方法前用 **throws** 声明抛出不就得了。异常的捕获就要做一些有意义的处理。

五、运行时异常和受检查异常

Exception 类可以分为两种：运行时异常和受检查异常。

1、运行时异常

RuntimeException 类及其子类都被称为运行时异常，这种异常的特点是 **Java** 编译器不去检查它，也就是说，当程序中可能出现这类异常时，即使没有用 **try...catch** 语句捕获它，也没有用 **throws** 语句声明抛出它，还是会编译通过。例如，当除数为零时，就会抛出 **java.lang.ArithmeticException** 异常。

2、受检查异常

除了 **RuntimeException** 类及其子类外，其他的 **Exception** 类及其子类都属于受检查异常，这种异常的特点是要么用 **try...catch** 捕获处理，要么用 **throws** 语句声明抛出，否则编译不会通过。

3、两者的区别

运行时异常表示无法让程序恢复运行的异常，导致这种异常的原因通常是由于执行了错误的操作。一旦出现错误，建议让程序终止。

受检查异常表示程序可以处理的异常。如果抛出异常的方法本身不处理或者不能处理它，那么方法的调用者就必须去处理该异常，否则调用会出错，连编译也无法通过。当然，这两种异常都是可以通过程序来捕获并处理的，比如除数为零的运行时异常：

```
public class HelloWorld {  
    public static void main(String[] args) {
```

```
System.out.println("Hello World!!!");
try{
    System.out.println(1/0);
}catch(ArithmeticException e){
    System.out.println("除数为 0!");
}
System.out.println("除数为零后程序没有终止啊，呵呵!!!");
}
}
```

运行结果：

Hello World!!!

除数为 0!

除数为零后程序没有终止啊，呵呵!!!

4、运行时错误

Error 类及其子类表示运行时错误，通常是由 **Java** 虚拟机抛出的，**JDK** 中与定义了一些错误类，比如 **VirtualMachineError**

和 **OutOfMemoryError**，程序本身无法修复这些错误。一般不去扩展 **Error** 类来创建用户自定义的错误类。而 **RuntimeException** 类表示程序代码中的错误，是可扩展的，用户可以创建特定运行时异常类。

Error(运行时错误)和运行时异常的相同之处是:Java 编译器都不去检查它们，当程序运行时出现它们，都会终止运行。

5、最佳解决方案

对于运行时异常，我们不要用 **try...catch** 来捕获处理，而是在程序开发调试阶段，尽量去避免这种异常，一旦发现该异常，正确的做法就会改进程序设计的代码和实现方式，修改程序中的错误，从而避免这种异常。捕获并

处理运行时异常是好的解决办法，因为可以通过改进代码实现来避免该种异常的发生。

对于受检查异常，没说的，老老实实去按照异常处理的方法去处理，要么用 **try...catch** 捕获并解决，要么用 **throws** 抛出！

对于 **Error**(运行时错误)，不需要在程序中做任何处理，出现问题后，应该在程序在外的地方找问题，然后解决。

六、异常转型和异常链

异常转型在上面已经提到过了，实际上就是捕获到异常后，将异常以新的类型的异常再抛出，这样做一般为了异常的信息更直观！比如：

```
public void run() throws MyException{
    ...
    try{
        ...
    }catch(IOException e){
        ...
        throw new MyException();
    }finally{
        ...
    }
}
```

异常链，在 **JDK1.4** 以后版本中，**Throwable** 类支持异常链机制。

Throwable 包含了其线程创建时线程执行堆栈的快照。它还包含了给出有关错误更多信息的消息字符串。最后，它还可以包含 **cause(原因)**：另一个导致此 **throwable** 抛出的 **throwable**。它也称为异常链 设施，因为 **cause** 自身也会有 **cause**，依此类推，就形成了异常链，每个异常都是由另一个异常引起的。

通俗的说，异常链就是把原始的异常包装为新的异常类，并在新的异常类中封装了原始异常类，这样做的目的在于找到异常的根本原因。

通过 **Throwable** 的两个构造方法可以创建自定义的包含异常原因的异常类型：

Throwable(String message, Throwable cause)

构造一个带指定详细消息和 **cause** 的新 **throwable**。

Throwable(Throwable cause)

构造一个带指定 **cause** 和 **(cause==null ? null : cause.toString())**(它通常包含类和 **cause** 的详细消息)的详细消息的新 **throwable**。

getCause()

返回此 **throwable** 的 **cause**;如果 **cause** 不存在或未知，则返回 **null**。

initCause(Throwable cause)

将此 **throwable** 的 **cause** 初始化为指定值。

在 **Throwable** 的子类 **Exception** 中，也有类似的指定异常原因的构造方法：

Exception(String message, Throwable cause)

构造带指定详细消息和原因的新异常。

Exception(Throwable cause)

根据指定的原因和 **(cause==null ? null : cause.toString())** 的详细消息构造新异常(它通常包含 **cause** 的类和详细消息)。

因此，可以通过扩展 **Exception** 类来构造带有异常原因的新的异常类。

七、Java 异常处理的原则和技巧

1、避免过大的 **try** 块，不要把不会出现异常的代码放到 **try** 块里面，尽量保持一个 **try** 块对应一个或多个异常。

2、细化异常的类型，不要不管什么类型的异常都写成 `Excetpion`。

3、`catch` 块尽量保持一个块捕获一类异常，不要忽略捕获的异常，捕获到后要么处理，要么转译，要么重新抛出新类型的异常。

4、不要把自己能处理的异常抛给别人。

5、不要用 `try...catch` 参与控制程序流程，异常控制的根本目的是处理程序的非正常情况。

```
<!--  
-->
```

（十三）java 网络通信编程

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

首先声明一下，刚开始学习 java 网络通信编程就对他有一种畏惧感，因为自己对网络一窍不通，所以。。。呵呵。。你懂得，昨天又仔细的学习了一遍，感觉其实 java 网络编程也没想象的那么难，不信，咱一起看看。。。呵呵。。

网络编程就是在两个或两个以上的设备(例如计算机)之间传输数据。程序员所作的事情就是把数据发送到指定的位置，或者接收到指定的数据，这个就是狭义的网络编程范畴。在发送和接收数据时，大部分的程序设计语言都设计了专门的 API 实现这些功能，程序员只需要调用即可。所以，基础的网络编程可以和打电话一样简单

一：首先看一下网络通讯的两种方式

1.TCP(传输控制协议)方式

TCP 方式就类似于拨打电话，使用该种方式进行网络通讯时，需要建立专门的虚拟连接，然后进行可靠的数据传输，如果数据发送失败，则客户端会自动重发该数据

2. UDP(用户数据报协议)方式

UDP 方式就类似于发送短信，使用这种方式进行网络通讯时，不需要建立专门的虚拟连接，传输也不是很可靠，如果发送失败则客户端无法获得这两种传输方式都是实际的网络编程中进行使用，重要的数据一般使用 TCP 方式进行数据传输，而大量的非核心数据则都通过 UDP 方式进行传递，在

一些程序中甚至结合使用这两种方式进行数据的传递。由于 TCP 需要建立专用的虚拟连接以及确认传输是否正确，所以使用 TCP 方式的速度稍微慢一些，而且传输时产生的数据量要比 UDP 稍微大一些。

总结一下 UDP 和 TCP 协议的区别

- 使用 UDP 时，每个数据报中都给出了完整的地址信息，因此无需要建立发送方和接收方的连接。

- 对于 TCP 协议，由于它是一个面向连接的协议，在 socket 之间进行数据传输之前必然要建立连接，所以在 TCP 中多了一个连接建立的时间

- 使用 UDP 传输数据时是有大小限制的，每个被传输的数据报必须限定在 64KB 之内。

- TCP 没有这方面的限制，一旦连接建立起来，双方的 socket 就可以按统一的格式传输大量的数据。

- UDP 是一个不可靠的协议，发送方所发送的数据报并不一定以相同的次序到达接收方。

- TCP 是一个可靠的协议，它确保接收方完全正确地获取发送方所发送的全部数据

- TCP 在网络通信上有极强的生命力，例如远程连接（Telnet）和文件传输（FTP）都需要不定长度的数据被可靠地传输。

- 相比之下 UDP 操作简单，而且仅需要较少的监护，因此通常用于局域网高可靠性的分散系统中 client/server 应用程序

二：基于 url 的网络编程

1.创建一个 URL

为了表示 URL，java.net 中实现了类 URL。我们可以通过下面的构造方法来初始化一个 URL 对象：

```
(1) public URL (String spec);  
    通过一个表示 URL 地址的字符串可以构造一个 URL 对象  
    URL urlBase=new URL("http://www. 263.net/")  
(2) public URL(URL context, String spec);  
    通过基 URL 和相对 URL 构造一个 URL 对象。  
    URL net263=new URL ("http://www.263.net/");  
    URL index263=new URL(net263, "index.html")  
(3) public URL(String protocol, String host, String file);  
    new URL("http", "www.gamelan.com", "/pages/Gamelan.net. html");  
(4) public URL(String protocol, String host, int port, String file);  
    URL gamelan=new URL("http", "www.gamelan.com", 80, "Pages/Gamelan.network.  
html");
```

注意：类 URL 的构造方法都声明抛弃非运行时例外

(MalformedURLException)，因此生成 URL 对象时，我们必须要对这一例外进行处理，通常是用 try-catch 语句进行捕获。格式如下：

```
try{  
    URL myURL= new URL(...)  
}catch (MalformedURLException e){  
    ...  
}
```

2. 解析一个 URL

一个 URL 对象生成后，其属性是不能被改变的，但是我们可以通过类 URL 所提供的方法来获取这些属性：

```
public String getProtocol() 获取该 URL 的协议名。  
public String getHost() 获取该 URL 的主机名。  
public int getPort() 获取该 URL 的端口号，如果没有设置端口，返回-1。  
public String getFile() 获取该 URL 的文件名。  
public String getRef() 获取该 URL 在文件中的相对位置。
```

`public String getQuery()` 获取该 URL 的查询信息。

`public String getPath()` 获取该 URL 的路径

`public String getAuthority()` 获取该 URL 的权限信息

`public String getUserInfo()` 获得使用者的信息

`public String getRef()`获得该 URL 的锚

3.从 URL 读取 WWW 网络资源

当我们得到一个 URL 对象后，就可以通过它读取指定的 WWW 资源。这时

我们将使用 URL 的方法 `openStream()`，其定义为：

`InputStream openStream();`

方法 `openStream()`与指定的 URL 建立连接并返回 `InputStream` 类的对象

以从这一连接中读取数据。

```
URL url = new URL("http://www.baidu.com");
```

```
//使用 openStream 得到一输入流并由此构造一个 BufferedReader 对象
```

```
BufferedReader br = new BufferedReader(new InputStreamReader( url.openStream()));
```

```
String line = null;
```

```
while(null != (line = br.readLine()))
```

```
{
```

```
System.out.println(line);
```

```
}
```

```
br.close();
```

三：客户端网络编程步骤

按照前面的基础知识介绍，无论使用 TCP 方式还是 UDP 方式进行网络通讯，

网络编程都是由客户端和服务端组成

1.客户端网络编程步骤

客户端(Client)是指网络编程中首先发起连接的程序,客户端一般实现程序界面和基本逻辑实现,在进行实际的客户端编程时,无论客户端复杂还是简单,以及客户端实现的方式,客户端的编程主要由三个步骤实现:

1、 建立网络连接

客户端网络编程的第一步都是建立网络连接。在建立网络连接时需要指定连接到的服务器的 IP 地址和端口号,建立完成以后,会形成一条虚拟的连接,后续的操作就可以通过该连接实现数据交换了。

2、 交换数据

连接建立以后,就可以通过这个连接交换数据了。交换数据严格按照请求响应模型进行,由客户端发送一个请求数据到服务器,服务器反馈一个响应数据给客户端,如果客户端不发送请求则服务器端就不响应。

根据逻辑需要,可以多次交换数据,但是还是必须遵循请求响应模型。

3、 关闭网络连接

在数据交换完成以后,关闭网络连接,释放程序占用的端口、内存等系统资源,结束网络编程。

最基本的步骤一般都是这三个步骤,在实际实现时,步骤 2 会出现重复,在进行代码组织时,由于网络编程是比较耗时的操作,所以一般开启专门的现场进行网络通讯。

2.服务器端网络编程步骤

服务器端(Server)是指在网络编程中被动等待连接的程序,服务器端一般实现程序的核心逻辑以及数据存储等核心功能。服务器端的编程步骤和客户端不同,是由四个步骤实现,依次是:

1、 监听端口

服务器端属于被动等待连接，所以服务器端启动以后，不需要发起连接，而只需要监听本地计算机的某个固定端口即可。

这个端口就是服务器端开放给客户端的端口，服务器端程序运行的本地计算机的 IP 地址就是服务器端程序的 IP 地址。

2、 获得连接

当客户端连接到服务器端时，服务器端就可以获得一个连接，这个连接包含客户端的信息，例如客户端 IP 地址等等，服务器端和客户端也通过该连接进行数据交换。

一般在服务器端编程中，当获得连接时，需要开启专门的线程处理该连接，每个连接都由独立的线程实现。

3、 交换数据

服务器端通过获得的连接进行数据交换。服务器端的数据交换步骤是首先接收客户端发送过来的数据，然后进行逻辑处理，再把处理以后的结果数据发送给客户端。简单来说，就是先接收再发送，这个和客户端的数据交换顺序不同。

其实，服务器端获得的连接和客户端连接是一样的，只是数据交换的步骤不同。

当然，服务器端的数据交换也是可以多次进行的。

在数据交换完成以后，关闭和客户端的连接。

4、 关闭连接

当服务器程序关闭时，需要关闭服务器端，通过关闭服务器端使得服务器监听的端口以及占用的内存可以释放出来，实现了连接的关闭。

四：一个基础的网络类——**InetAddress** 类

该类的功能是代表一个 IP 地址，并且将 IP 地址和域名相关的操作方法包含在该类的内部。

关于该类的使用，下面通过一个基础的代码示例演示该类的使用，代码如下：

[java] [view plaincopyprint?](#)

```
1. <SPAN style="BACKGROUND-COLOR: #333333">public class InetAddressDemo
   {
2.
3. public static void main(String[] args) {
4.
5. try {
6.
7. // 使用域名创建对象
8.
9. InetAddress inet1 = InetAddress.getByName("www.163.com");
10.
11.System.out.println(inet1);
12.
13.// 使用 IP 创建对象
14.
15.InetAddress inet2 = InetAddress.getByName("127.0.0.1");
16.
17.System.out.println(inet2);
18.
19.// 获得本机地址对象
20.
21.InetAddress inet3 = InetAddress.getLocalHost();
```

```

22.
23. System.out.println(inet3);
24.
25. // 获得对象中存储的域名
26.
27. String host = inet3.getHostName();
28.
29. System.out.println("域名: " + host);
30.
31. // 获得对象中存储的 IP
32.
33. String ip = inet3.getHostAddress();
34.
35. System.out.println("IP:" + ip);
36.
37. } catch (Exception e) {
38. }
39. }
40. }
41. </SPAN>

```

注：InetAddress 类没有明显的构造函数。为生成一个 InetAddress 对象，必须运用一个可用的工厂方法。

–工厂方法（factory method）仅是一个类中静态方法返回一个该类实例的约定。对于 InetAddress，三个方法 getLocalHost()、getByName()以及 getAllByName()可以用来创建 InetAddress 的实例

- 如果这些方法不能解析主机名，它们引发一个 UnknownHostException 异常。

五：TCP 编程

在 Java 语言中，对于 TCP 方式的网络编程提供了良好的支持，在实际实现时，以 `java.net.Socket` 类代表客户端连接，以 `java.net.ServerSocket` 类代表服务器端连接。在进行网络编程时，底层网络通讯的细节已经实现了比较高的封装，所以在程序员实际编程时，只需要指定 IP 地址和端口号码就可以建立连接了。

在客户端网络编程中，首先需要建立连接，在 Java API 中以 `java.net.Socket` 类的对象代表网络连接

客户端

1) 建立 Socket 连接

```
Socket socket2 = new Socket("www.sohu.com",80);
```

2) 按照“请求-响应”模型进行网络数据交换

在 Java 语言中，数据传输功能由 Java IO 实现，也就是说只需要从连接中获得输入流和输出流即可，然后将需要发送的数据写入连接对象的输出流中，在发送完成以后从输入流中读取数据即可。示例代码如下：

```
OutputStream os = socket1.getOutputStream(); //获得输出流
```

```
InputStream is = socket1.getInputStream(); //获得输入流
```

这里获得的只是最基本的输出流和输入流对象，还可以根据前面学习到的 IO 知识，使用流的嵌套将这些获得到的基本流对象转换成需要的装饰流对象，从而方便数据的操作。

3) 关闭网络连接

```
socket1.close();
```

服务器端

首先需要说明的是，客户端的步骤和服务端端的编写步骤不同，所以在学习服务器端编程时注意不要和客户端混淆起来。

1) 监听端口

```
ServerSocket ss = new ServerSocket(10000);
```

2) 获得连接

当有客户端连接到达时，建立一个和客户端连接对应的 **Socket** 连接对象，从而释放客户端连接对于服务器端端口的占用

```
Socket socket = ss.accept();
```

该代码实现的功能是获得当前连接到服务器端的客户端连接。需要说明的是 **accept** 和前面 IO 部分介绍的 **read** 方法一样，都是一个阻塞方法，也就是当无连接时，该方法将阻塞程序的执行，直到连接到达时才执行该行代码。另外获得的连接会在服务器端的该端口注册，这样以后就可以通过在服务器端的注册信息直接通信，而注册以后服务器端的端口就被释放出来，又可以继续接受其它的连接了。

3) 按照“请求-响应”模型进行网络数据交换

这里获得的 **Socket** 类型的连接就和客户端的网络连接一样了，只是服务器端需要首先读取发送过来的数据，然后进行逻辑处理以后再发送给客户端，也就是交换数据的顺序和客户端交换数据的步骤刚好相反

```
InputStream is = ss.getInputStream(); //获得输入流
```

```
OutputStream os = ss.getOutputStream(); //获得输出流
```

4) 关闭服务器端连接

```
ss.close();
```

以上就是基本的 TCP 类型的服务器和客户端代码实现的步骤，下面以一个简单的 echo（回声）服务实现为例子，介绍综合使用示例，实现的代码如下：

[java] [view plaincopyprint?](#)

```
1. public class Constants {
2.
3.     public static void main(String[] args) {
4.         ServerSocket serverSocket = null;
5.         Socket socket = null;
6.         OutputStream os = null;
7.         InputStream is = null;
8.         // 监听端口号
9.         int port = 10000;
10.        try {
11.            // 建立连接
12.            serverSocket = new ServerSocket(port);
13.            // 获得连接
14.            socket = serverSocket.accept();
15.            // 接收客户端发送内容
16.            is = socket.getInputStream();
17.            byte[] b = new byte[1024];
18.            int n = is.read(b);
19.            // 输出
20.            System.out.println("客户端发送内容为: " + new String(b, 0, n));
21.            // 向客户端发送反馈内容
22.            os = socket.getOutputStream();
23.            os.write(b, 0, n);
24.        } catch (Exception e) {
25.            e.printStackTrace();
26.        } finally {
27.            try {
28.                // 关闭流和连接
29.                os.close();
30.                is.close();
```

```
31.         socket.close();
32.         serverSocket.close();
33.     } catch (Exception e) {
34.     }
35. }
36. }
37. }
```

UDP 编程

UDP(User Datagram Protocol)，中文意思是用户数据报协议使用该种方式无需建立专用的虚拟连接，由于无需建立专用的连接，所以对于服务器的压力要比 TCP 小很多，所以也是一种常见的网络编程方式。但是使用该种方式最大的不足是传输不可靠，当然也不是说经常丢失，就像大家发短信息一样，理论上存在收不到的可能

在 Java API 中，实现 UDP 方式的编程，包含客户端网络编程和服务端网络编程，主要由两个类实现，分别是：

I DatagramSocket

DatagramSocket 类实现“网络连接”，包括客户端网络连接和服务端网络连接。虽然 UDP 方式的网络通讯不需要建立专用的网络连接，但是毕竟还是需要发送和接收数据，DatagramSocket 实现的就是发送数据时的发射器，以及接收数据时的监听器的角色。类比于 TCP 中的网络连接，该类既可以用于实现客户端连接，也可以用于实现服务器端连接。

I DatagramPacket

DatagramPacket 类实现对于网络中传输的数据封装，也就是说，该类的对象代表网络中交换的数据。在 UDP 方式的网络编程中，无论是需要发送的数据还是需要接收的数据，都必须被处理成 DatagramPacket 类型的对象，

该对象中包含发送到的地址、发送到的端口号以及发送的内容等。其实 `DatagramPacket` 类的作用类似于现实中的信件，在信件中包含信件发送到的地址以及接收人，还有发送的内容等，邮局只需要按照地址传递即可。在接收数据时，接收到的数据也必须被处理成 `DatagramPacket` 类型的对象，在该对象中包含发送方的地址、端口号等信息，也包含数据的内容。和 TCP 方式的网络传输相比，IO 编程在 UDP 方式的网络编程中变得不是必须的内容，结构也要比 TCP 方式的网络编程简单一些。

UDP 客户端编程涉及的步骤也是 4 个部分：建立连接、发送数据、接收数据和关闭连接。

1) 建立连接:

```
DatagramSocket ds = new DatagramSocket();
```

该客户端连接使用系统随机分配的一个本地计算机的未用端口号

当然，可以通过制定连接使用的端口号来创建客户端连接。

```
DatagramSocket ds = new DatagramSocket(5000);
```

一般在建立客户端连接时没有必要指定端口号码。

2) 发送数据

在发送数据时，需要将需要发送的数据内容首先转换为 `byte` 数组，然后将数据内容、服务器 IP 和服务器端口号一起构造成一个 `DatagramPacket` 类型的对象，这样数据的准备就完成了了，发送时调用网络连接对象中的 `send` 方法发送该对象即可

代码示例:

[java] [view plaincopyprint?](#)

```

1. <SPAN style="BACKGROUND-COLOR: #666666; COLOR: #333333"> String s
   = "Hello";
2.   String host = "127.0.0.1";
3.   int port = 10001;
4.   //将发送的内容转换为 byte 数组
5.   byte[] b = s.getBytes();
6.   //将服务器 IP 转换为 InetAddress 对象
7.   InetAddress server = InetAddress.getByName(host);
8.   //构造发送的数据包对象
9.   DatagramPacket sendDp = new DatagramPacket(b,b.length,server,port);
10.  //发送数据
11.  ds.send(sendDp);</SPAN>

```

在该示例代码中，不管发送的数据内容是什么，都需要转换为 **byte** 数组，然后将服务器端的 IP 地址构造成 **InetAddress** 类型的对象，在准备完成以后，将这些信息构造成一个 **DatagramPacket** 类型的对象，在 UDP 编程中，发送的数据内容、服务器端的 IP 和端口号，都包含在 **DatagramPacket** 对象中。在准备完成以后，调用连接对象 **ds** 的 **send** 方法把 **DatagramPacket** 对象送出去即可。

3) UDP 客户端编程中接收数据

首先构造一个数据缓冲数组，该数组用于存储接收的服务器端反馈数据，该数组的长度必须大于或等于服务器端反馈的实际有效数据的长度。然后以该缓冲数组为基础构造一个 **DatagramPacket** 数据包对象，最后调用连接对象的 **receive** 方法接收数据即可。接收到的服务器端反馈数据存储在 **DatagramPacket** 类型的对象内部

示例代码：

[java] view plaincopyprint?

```
1. <SPAN style="BACKGROUND-COLOR: #666666"> //构造缓冲数组
2. byte[] data = new byte[1024];
3. //构造数据包对象
4. DatagramPacket receiveDp = new DatagramPacket(data,data.length);
5. //接收数据
6. ds.receive(receiveDp);
7. //输出数据内容
8. byte[] b = receiveDp.getData(); //获得缓冲数组
9. int len = receiveDp.getLength(); //获得有效数据长度
10. String s = new String(b,0,len);
11. System.out.println(s);</SPAN>
```

代码讲解：首先构造缓冲数组 **data**，这里设置的长度 **1024** 是预估的接收到的数据长度，要求该长度必须大于或等于接收到的数据长度，然后以该缓冲数组为基础，构造数据包对象，使用连接对象 **ds** 的 **receive** 方法接收反馈数据，由于在 **Java** 语言中，除 **String** 以外的其它对象都是按照地址传递，所以在 **receive** 方法内部可以改变数据包对象 **receiveDp** 的内容，这里的 **receiveDp** 的功能和返回值类似。数据接收到以后，只需要从数据包对象中读取出来就可以了，使用 **DatagramPacket** 对象中的 **getData** 方法可以获得数据包对象的缓冲区数组，但是缓冲区数组的长度一般大于有效数据的长度，换句话说，也就是缓冲区数组中只有一部分数据是反馈数据，所以需要使用 **DatagramPacket** 对象中的 **getLength** 方法获得有效数据的长度，则有效数据就是缓冲数组中的前有效数据长度个内容，这些才是真正的服务器端反馈的数据的内容

4) 关闭连接

```
ds.close();
```

UDP 方式服务器端网络编程

1) 首先 **UDP** 方式服务器端网络编程需要建立一个连接，该连接监听某个端口：

```
DatagramSocket ds = new DatagramSocket(10010);
```

由于服务器端的端口需要固定，所以一般在建立服务器端连接时，都指定端口号

2) 接收客户端发送过来的数据

其接收的方法和客户端接收的方法一样，其中 **receive** 方法的作用类似于 TCP 方式中 **accept** 方法的作用，该方法也是一个阻塞方法，其作用是接收数据。

```
ds.receive()
```

接收到客户端发送过来的数据以后，服务器端对该数据进行逻辑处理，然后将处理以后的结果再发送给客户端，在这里发送时就比客户端要麻烦一些，因为服务器端需要获得客户端的 IP 和客户端使用的端口号，这个都可以从接收到的数据包中获得。示例代码如下：

```
//获得客户端的 IP
```

```
InetAddress clientIP = receiveDp.getAddress();
```

```
//获得客户端的端口号
```

```
Int clientPort = receiveDp.getPort();
```

3) 关闭连接

```
ds.close()
```

好了，占时就总结到这吧，总结的不是很全面，但很基础，应该适合初学者学习，由于本人也是初学者的小菜鸟，所有很多东西可能都涉及不到，希望大家见谅！

（十四）Html 基础积累总结（上）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

注：由于本文内含有大量 **html** 标签，所以在排版上有些困难，所以排版有点难看，请大家见谅

一：首先看页面标记

1. html 文件结构

```
<HTML>
<HEAD>
    <title>, <base>, <link>, <isindex>, <meta>
</HEAD>
<BODY>
    HTML 文件的正文写在这里... ..
</BODY>
</HTML>
```

2. 语言字符集(Charsets)的信息

```
<meta http-equiv="Content-Type" content="text/html; charset=#">
#可以是 gbk, utf-8 等
```

3. 背景色彩和文字色彩

```
<body bgcolor=# text=# link=# alink=# vlink=#>
bgcolor --- 背景色彩
```

text --- 非可链接文字的色彩

link --- 可链接文字的色彩

alink --- 正被点击的可链接文字的色彩

vlink --- 已经点击(访问)过的可链接文字的色彩

#=rrggbb

色彩是用 16 进制的 红—绿—蓝(red-green-blue, RGB) 值来表示。

16 进制的数码有: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f.

背景图象 <body background="image-URL">

4.链接(Link)

基本语法 ...

这是一个

链接的例子。

点一下带下划线的文字!

跳转到页面的另外一个地方

 ...

 ...

跳转到下一个"链接点"<P>

下一个链接点

跳转到另一个页面的某个地方

 ...

 ...

跳转到另一个页面的某个地方。

开一个新的(浏览器)窗口 (Target Window)

 ...

开一个新窗口!

5.标尺线

<hr>

<hr>

<hr size=#>
<hr size=10>
<hr width=#>
<hr width=50>
<hr width=50%>
<hr align=#> #=left, right
<hr width=50% align=left>
<hr width=50% align=right>
<hr noshade>
<hr noshade>
<hr color=#>

#=rrggbb 16 进制 RGB 数码，或者是下列预定义色彩：

Black, Olive, Teal, Red, Blue, Maroon, Navy, Gray, Lime,
Fuchsia, White, Green, Purple, Silver, Yellow, Aqua
<hr color="red">

二：然后来看一下字体的设置

1.标题字体(Header)

<h#> ... </h#> #=1, 2, 3, 4, 5, 6
<h1>今天天气真好！</h1>今天天气真好！
<h2>今天天气真好！</h2>今天天气真好！
<h3>今天天气真好！</h3>今天天气真好！
<h4>今天天气真好！</h4>今天天气真好！
<h5>今天天气真好！</h5>今天天气真好！
<h6>今天天气真好！</h6>今天天气真好！

· <hn>---</hn> 这些标记显示黑体字。

· <hn>---</hn> 这些标记自动插入一个空行，不必用 <p> 标记再加空行。

因此在一行中无法使用不同大小的字体。

2.字体大小

 ... #=1, 2, 3, 4, 5, 6, 7 or +#, -#

<basefont size=#> #=1, 2, 3, 4, 5, 6, 7

今天天气真好！ 今天天气真好！

今天天气真好！ 今天天气真好！

今天天气真好！ 今天天气真好！

今天天气真好！ 今天天气真好！

3.物理字体(Physical Style)

今天天气真好！ 今天天气真好！

<i>今天天气真好！ </i> 今天天气真好！

<u>今天天气真好！ </u> 今天天气真好！

<tt>今天天气真好！ </tt> 今天天气真好！

^{今天天气真好！} 今天天气真好！

_{今天天气真好！} 今天天气真好！

<s>今天天气真好！ </s> 今天天气真好！

<strike>今天天气真好！ </strike> 今天天气真好！

4.逻辑字体(Logical Style)

今天天气真好！ 今天天气真好！

今天天气真好！ 今天天气真好！

<code>今天天气真好！ </code> 今天天气真好！

<var>今天天气真好！ </var> 今天天气真好！

<dfn>今天天气真好！ </dfn> 今天天气真好！

<cite>今天天气真好！ </cite> 今天天气真好！

<small>今天天气真好！ </small> 今天天气真好！

<big>今天天气真好！ **</big>** 今天天气真好！

5.指定“字体大小”的标记和“指定字体”的标记的组合使用

<i>

****今天**** 天气**** 真好！ ****

</i>

今天 天气真好！

6.字体颜色

指定颜色 ** ... **

#=rrggbb 16 进制数码，或者是下列预定义色彩：

Black, Olive, Teal, Red, Blue, Maroon, Navy, Gray, Lime,
Fuchsia, White, Green, Purple, Silver, Yellow, Aqua

****White**** &

****White****

7.客户端字体(Font Face)

** ... **

#=客户端可获得的字体（微软雅黑，roman 等）

**** Hellow World!****

Hellow World!

8.字符实体(Entities)

&#; **#=**字符实体名称 或者 **ascii** 值

HTML2.0 的字符集

&; **&**

<; **<**

>; **>**

"; **"**

HTML2.0 字符实体名称列表

HTML3.2 字符实体名称列表

ISO 字符实体名称列表

三：在看一下文字布局

1.行的控制

段(Paragraph) (可以看作是空行) `<p>`

你好吗？`<p>`很好。

你好吗？

很好。

换行 `
`

你好吗？`
`很好。

你好吗？

很好。

不换行`<nobr>`

`<nobr>`

请改变您浏览器窗口的宽度，使之小于这一行的宽度，看看这个标记的作用！

`</nobr>`

请改变您浏览器窗口的宽度，使之小于这一行的宽度，看看这个标记的作用！

2.文字的对齐(Alignment)

`<hn align=#>...</hn>`

`<p align=#>...</p>` #=left, center, right

`<h3 align=center>Hello</h3>`

`<h3 align=right>Hello</h3>`

Hello

Hello

`<center>...</center>`

`<center>Hello</center>`

Hello

3.文字的分区(Division)显示

<div align=left> ... </div>

<div align=center> ... </div>

Can you feel happiness without unpleasant?

Please show me your smile.

4.列表

无序列表 ...

Today

Tommorrow

· Today

· Tommorrow

有序列表 ...

Today

Tommorrow

1 Today

2 Tommorrow

定义列表(Definition lists) <dl><dt>...<dd>...</dl>

<dl>

<dt>Today

<dd>Today is yesterday.

<dt>Tomorrow

<dd>Tomorrow is today.

</dl>

Today

Today will be yesterday.

Tomorrow

Tomorrow will be today.

5.预格式化文本(Preformatted Text)

<pre>...</pre>

<pre>

Please use your card.

VISA Master

Here is an order form.

Fax

Air Mail

</pre>

Please use your card

VISA Master

Here is an order form.

· Fax

· Air Mail

<xmp>...</xmp>

<xmp>

Please use your card.

VISA Master

Here is an order form.

Fax

Air Mail

</xmp>

Please use your card.

VISA Master

Here is order form.

Fax

Air Mail

闪烁 <blink>...</blink>

<BLINK> 闪烁！ 闪烁！ </BLINK>

四：下一个看一下图象

1.链入图象的基本语法

 #=图象的 URL

#=在浏览器尚未完全读入图象时，在图象位置显示的文字。

2. 图象和文字的对齐

 #=top, middle, bottom

 My face!

My Face!

· 只有一行文字才可以放在图象的两边。（不知道翻译的对不对？）

· Only one text line can be flown into the both side of Image.

3. 图象在页面中的对齐/布局(Floating Image)

My Face!

It is always

smiling.

Hahaha....

My Face!

It is always

smiling.

Hahaha....

<br clear=all>

My Face!

It is always

<br clear=all>

smiling.

Hahaha....

My Face!

It is always

smiling.

Hahaha....

 #=value

My Face!

It is always

smiling.

Hahaha....

My Face!
It is always
smiling.
Hahaha....

4.边框

```
<img border=#> #=value  
<a href="URL">  
<img src=URL border=15>  
</a>
```

5.客户端图象映射图

示例代码:

```
  
<map name="Face">  
<!Text BOTTON>  
  <area shape="rect" href="page.html" coords="140,20,280,60">  
<!Triangle BOTTON>  
  <area shape="poly"  
href="image.html"      coords="100,100,180,80,200,140"> <!FACE>  
  <area shape="circle"      href="new.html"      coords="80,100,60">  
</map>
```

五：下面看一下常用的表单

1.基本语法

```
<form action="url" method=*>  
...  
...  
<input type=submit> <input type=reset>  
</form>  
*=GET, POST
```

表单中提供给用户的输入形式

`<input type=* name=**>`

`*`=text,文本框 password 密码框, checkbox 复选框, radio 单选框, image 图象

坐标, hidden 隐藏表单, submit 提交, reset 重置

`**`=Symbolic Name for CGI script

2.列表框(Selectable Menu)

基本语法

`<select name=*>`

`<option> ...`

`</select>`

`<option selected>`

`<option value=**>`

`<select name=fruits>`

`<option>Banana`

`<option selected>Apple`

`<option value=My_Favorite>Orange`

`</select><p>`

`<option value=** size=3>`

`<select size=** multiple>`

注意,是用 Ctrl 键配合鼠标实现多选。

(和 MS-WINDOWS 的 File Manager 一样)

3.文本域

`<textarea name=* rows=** cols=**> ... </textarea>`

对于很长的行是否进行换行的设置(Word Wrapping)

`<textarea wrap=off> ... </textarea>`

不换行,是缺省设置。

`<textarea wrap=soft> ... </textarea>`

“软换行”,好象 MS—WORD 里的“软回车”。

`<textarea wrap=hard> ... </textarea>`

“硬换行”,好象 MS—WORD 里的“硬回车”。

（十五）Html 基础积累总结（下） .

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

一：表格

1.表格的基本语法

`<table>...</table>` - 定义表格

`<tr>` - 定义表行

`<th>` - 定义表头

`<td>` - 定义表元(表格的具体数据)

带边框的表格:

`<table border>`

`<tr><th>Food</th><th>Drink</th><th>Sweet</th>`

`<tr><td>A</td><td>B</td><td>C</td>`

`</table>`

不带边框的表格:

`<table>`

`<tr><th>Food</th><th>Drink</th><th>Sweet</th>`

`<tr><td>A</td><td>B</td><td>C</td>`

`</table>`

2.跨多行、多列的表元(Table Span)

跨多列的表元 `<th colspan=#>`

`<table border>`

`<tr><th colspan=3> Morning Menu</th>`

`<tr><th>Food</th> <th>Drink</th> <th>Sweet</th>`

`<tr><td>A</td><td>B</td><td>C</td>`

`</table>`

跨多行的表元 `<th rowspan=#>`

`<table border>`

`<tr><th rowspan=3> Morning Menu</th>`

`<th>Food</th> <td>A</td></tr>`

`<tr><th>Drink</th> <td>B</td></tr>`

`<tr><th>Sweet</th> <td>C</td></tr>`

`</table>`

3.表格尺寸设置

边框尺寸设置:

`<table border=#>`

表格尺寸设置:

`<table border width=# height=#>`

表元间隙设置:

<table border cellspacing=#>

表元内部空白设置:

<table border cellpadding=#>

4.表格内文字的对齐/布局

<tr align=#>

<th align=#> #=left, center, right

<td align=#>

<tr valign=#>

<th valign=#> #=top, middle, bottom, baseline

<td valign=#>

5.表格在页面中的对齐/布局(Floating Table)

<table align=left>表格在页面靠左

<table align=right>

<table vspace=# hspace=#> #space value

<caption align=#> ... </caption> #=left, center, right

6.表格的标题

<table border>

<caption align=center>Lunch</caption>

<tr><th>Food</th><th>Drink</th><th>Sweet</th>

<tr><td>A</td><td>B</td><td>C</td>

</table>此表格的标题在页面中间

<caption valign=#> ... </caption> #=top, bottom

7.表格的色彩

表元的背景色彩和背景图象

<th bgcolor=#>

<th background="URL">

#=rrggbb 16 进制 RGB 数码, 或者是下列预定义色彩名称:

Black, Olive, Teal, Red, Blue, Maroon, Navy, Gray, Lime,

Fuchsia, White, Green, Purple, Silver, Yellow, Aqua

表格边框的色彩

<table bordercolor=#>

表格边框色彩的亮度控制

`<table bordercolorlight=#>`

`<table bordercolordark=#>`

`<table cellpadding=5 border=5 bordercolorlight=White bordercolordark=Maroon>`

`<tr><th>Food</th><th>Drink</th><th>Sweet</th>`

`<tr><td>A</td><td>B</td><td>C</td>`

`</table>`

8.表格的分组显示(Structured Table)

按行分组

`<thead> ... </thead>` - 表的题头(Header)

`<tbody> ... </tbody>` - 表的正文(Body)

`<tfoot> ... </tfoot>` - 表的脚注(Footer)

`<table border>`

`<thead>`

`<tr><th>Food</th><th>Drink</th><th>Sweet</th>`

`</thead>`

`<tbody>`

`<tr><td>A</td><td>B</td><td>C</td>`

`<tr><td>D</td><td>E</td><td>F</td>`

`</tbody>`

`</table>`

按列分组

`<colgroup align=#> #=left, right, center`

列的属性控制

`<col span=#> #=从左数起, 具有指定属性的列的列数`

`<col align=#> #=left, right, center`

9.表格中边框的显示

显示所有 4 个边框 `<table frame=box>`

只显示上边框 `<table frame=above>`

只显示下边框 `<table frame=below>`

只显示上、下边框 `<table frame=hsides>`

只显示左、右边框 `<table frame=vsides>`

只显示左边框 `<table frame=lhs>`

只显示右边框 `<table frame=rhs>`

不显示任何边框 `<table frame=void>`

10.表格中分隔线(Rules)的显示

显示所有分隔线 <table rules=all>

只显示组(Groups)与组之间的分隔线 <table rules=groups>

只显示行与行之间的分隔线 <table rules=rows>

只显示列与列之间的分隔线 <table rules=cols>

不显示任何分隔线 <table rules=none>

二：多窗口页面

1.基本语法

<frameset> ... </frameset>

<frame src="url">

<noframes> ... </noframes>

在 <noframes> 标记后的文字将只出现在不支持 FRAMES 的浏览器中。

<HTML>

<HEAD>

</HEAD>

<FRAMESET>

<FRAME SRC="url">

<NOFRAMES> ... </NOFRAMES>

</FRAMESET>

</HTML>

2.各窗口的尺寸设置

<frameset cols=#>

纵向排列多个窗口：

<frameset cols=30%,20%,50%>

<frame src="A.html">

<frame src="B.html">

<frame src="C.html">

</frameset>

<frameset rows=#>

横向排列多个窗口：

<frameset rows=25%,25%,50%>

<frame src="A.html">

<frame src="B.html">

```
<frame src="C.html">
```

```
</frameset>
```

COLS & ROWS

纵横排列多个窗口:

```
<frameset cols=20%,*>
```

```
<frame src="A.html">
```

```
    <frameset rows=40%,*>
```

```
        <frame src="B.html">
```

```
        <frame src="C.html">
```

```
    </frameset>
```

```
</frameset>
```

不允许各窗口改变大小 <frame noresize>

缺省设置是允许各窗口改变大小的。

3.各窗口间相互操作(Frame Target)

窗口标识(Frame Name)

```
<frame name=#>
```

```
<a href=url target=#>
```

```
<frameset cols=50%,50%>
```

```
<frame src="A.html">
```

```
<frame src="B.html" name="HELLO">
```

```
</frameset>
```

特殊的 4 类操作(很有用喔)

```
<a href=url target=_blank> 新窗口
```

```
<a href=url target=_self> 本窗口
```

```
<a href=url target=_parent> 父窗口
```

```
<a href=url target=_top> 整个浏览器窗口
```

4.Frame 的外观(Appearance)

各窗口边框的设置 <frame frameborder=#> #=yes, no / 1, 0

```
<frameset rows=30%,*>
```

```
<frame src="Acol.html" frameborder=1>
```

```
<frameset cols=30%,*>
```

```
    <frame src="Bcol.html" frameborder=0>
```

```
    <frame src="Ccol.html" frameborder=0>
```

```
</frameset>
```

</frameset>

各窗口间空白区域的设置

<frameset framespacing=#> #=空白区域的大小

边框色彩 <frameset bordercolor=#>

页面空白(Margin) <frame marginwidth=# marginheight=#>

卷滚条设置 <frame scrolling=#> #=yes, no, auto

浮动窗口(Floating Frame)

<iframe src=# name=##> ... </iframe>

#=初始页面的 URL

##=窗口标识(Frame Name)(之后可对此标识进行各窗口间相互操作)

... = 此处文字将只出现在不支持 FRAMES 的浏览器中。

三：会移动的文字

1.基本语法

<marquee> ... </marquee>

<marquee>啦啦啦，我会移动耶！ </marquee>

2.文字移动属性的设置

方向 <direction=#> #=left, right

<marquee direction=left>啦啦啦，我从右向左移！ </marquee> <P>

<marquee direction=right>啦啦啦，我从左向右移！ </marquee>

方式 <behavior=#> #=scroll, slide, alternate

<marquee behavior=scroll>啦啦啦，我一圈一圈绕着走！ </marquee> <P>

循环 <loop=#> #=次数；若未指定则循环不止(infinite)

<marquee loop=3 width=50% behavior=scroll>啦啦啦，我只走 3 趟哟！

</marquee> <P>

速度 <scrollamount=#>

<marquee scrollamount=20>啦啦啦，我走得好快哟！ </marquee>

延时 <scrolldelay=#>

<marquee scrolldelay=500 scrollamount=100>啦啦啦，我走一步，停一停！ </marquee>

3.外观(Layout)设置

对齐方式(Align) <align=#> #=top, middle, bottom

<marquee align=# width=400>啦啦啦，我会移动耶！ </marquee>

对齐上沿、中间、下沿。

底色 <bgcolor=#>

<marquee bgcolor=aaaaee>啦啦啦，我会移动耶！</marquee>

面积 <height=# width=#>

<marquee height=40 width=50% bgcolor=aaeeaa>

啦啦啦，我会移动耶！

</marquee>

四：多媒体的嵌入

1.嵌入多媒体文本(EMBED)

基本语法 <embed src=#> #=URL

本标记可以用来在主页中嵌入多媒体文本，如：

电影(movie), 声音(sound), 虚拟现实语言(vrml)... ..

体会 <embed> 标记，您需要把 plugin 安装完备。

请注意：embed attributes are different between each plugins

2.背景音乐

<bgsound src=#> #=WAV 文件的 URL

<bgsound loop=#> #=循环数

<bgsound src="sound.wav" loop=3>

3.插入视频剪辑

用 url.avi 这一 AVI(Video for MS—WINDOWS) 文件来播放视频；

用 url.gif 这一 GIF 图象作为视频的封面，即：在浏览器

尚未完全读入 AVI 文件时，先在 AVI 播放区域显示该图象。

何时开始播放 AVI #=fileopen, mouseover

缺省值是 #=fileopen，即在链接到含本标记的页面(如本页)时开始播放 AVI。

mouseover 是指您把鼠标移到 AVI 播放区域之上时才开始播放 AVI。

也可以两者同时设置：

另外，用鼠标在 AVI 播放区域点击一下，也将令浏览器开始播放该 AVI。

控制条

用来在视频窗口下附加 MS—WINDOWS 的 AVI 播放控制条。

循环播放 ****

<loop=infinite> 将循环播放不止。

延时 **** #=毫秒数

用 CSS 的话，我们可以这样获得同样的效果：

```
body {background-color: #FF0000;}
```

上例也向你展示了基本的 CSS 模型：

为 HTML 文档应用 CSS，有三种方法可供选择。下面对这三种方法进行了概括。我们建议你对第三种方法（即外部样式表）予以关注。

方法 1：行内样式表（**style** 属性）

为 HTML 应用 CSS 的一种方法是使用 HTML 属性 **style**。我们在上例的基础之上，通过行内样式表将页面背景设为红色：

```
<body style="background-color: #FF0000;">
```

方法 2：内部样式表（**style** 元素）

为 HTML 应用 CSS 的另一种方法是采用 HTML 元素 **style**。比如像这样：

```
<style type="text/css">
  body {background-color: #FF0000;}
</style>
```

方法 3：外部样式表（引用一个样式表文件）

我们推荐采用这种引用外部样式表的方法。外部样式表就是一个扩展名为 **css** 的文本文件。跟其他文件一样，你可以把样式表文件放在 **Web** 服务器上或者本地硬盘上。

导入语法：

```
<link rel="stylesheet" type="text/css" href="style/style.css" />
```

注意要在 **href** 属性里给出样式表文件的地址。

这行代码必须被插入 HTML 代码的头部(header)，即放在标签 **<head>** 和标签 **</head>** 之间。就像这样：

```
<html>
  <head>
    <title>我的文档</title>
    <link rel="stylesheet" type="text/css" href="style/style.css" />
  </head>
  <body>
    ...
```

这个链接告诉浏览器：在显示该 HTML 文件时，应使用给出的 CSS 文件进行布局。

这种方法的优越之处在于：多个 HTML 文档可以同时引用一个样式表。换句话说，可以用一个 CSS 文件来控制多个 HTML 文档的布局。

这一方法可以令你省去许多工作。例如，假设你要修改某网站的所有网页（比方说有 100 个网页）的背景颜色，采用外部样式表可以避免你手工一一修改这 100 个 HTML

文档的工作。采用外部样式表，这样的修改只需几秒钟即可搞定——修改外部样式表文件里的代码即可。

然后，把这两个文件放在同一目录下。记得在保存文件时使用正确的扩展名（分别为“htm”和“css”）。

二：颜色与背景

前台页面背景和颜色主要包括下面内容：

- color
- background-color
- background-image
- background-repeat
- background-attachment
- background-position
- background

前景色：‘color’属性

CSS 属性 color 用于指定元素的前景色。

例如，假设你要让页面中的所有标题（headline）都显示为深红色，而这些标题采用的都是 h1 元素，那么可以用下面的代码来实现把 h1 元素的前景色设为红色。

```
h1 {  
    color: #ff0000;  
}
```

CSS 属性 background-color 用于指定元素的背景色。

因为 body 元素包含了 HTML 文档的所有内容，所以，如果要改变整个页面的背景色的话，那么为 body 元素应用 background-color 属性就行了。

你也可以为其他包含标题或文本的元素单独应用背景色。在下例中，我们为 body 和 h1 元素分别应用了不同的背景色。

```
body {  
    background-color: #FFCC66;  
}  
h1 {  
    color: #990000;  
    background-color: #FC9804;  
}
```

注意：我们为 h1 元素应用了两个 CSS 属性，它们之间以分号（“;”）分隔。

CSS 属性 background-image 用于设置背景图像。

如果要把这个蝴蝶的图片作为网页的背景图像，只要在 **body** 元素上应用 **background-image** 属性、然后给出蝴蝶图片的存放位置就行了。

```
body {
  background-color: #FFCC66;
  background-image: url("butterfly.gif");
}
h1 {
  color: #990000;
  background-color: #FC9804;
}
```

注意我们指定图片存放位置的方式：**url("butterfly.gif")**。这表明图片文件和样式表存放在同一目录下。你也可以引用存放在其他目录的图片，只需给出存放路径即可（比如 **url("../images/butterfly.gif")**）；此外，你甚至可以通过给出图片的地址来引用因特网（Internet）上的图片（比如 **url("http://www.html.net/butterfly.gif")**）。

平铺背景图像[**background-repeat**]

下表概括了 **background-repeat** 的四种不同取值。

值	描述	示例
background-repeat:repeat-x	图像横向平铺	显示示例
background-repeat:repeat-y	图像纵向平铺	显示示例
background-repeat:repeat	图像横向和纵向都平铺	显示示例
background-repeat:no-repeat	图像不平铺	显示示例

例如，为了避免平铺背景图像，代码应该这样：

```
body {
  background-image: url("butterfly.gif");
  background-repeat: no-repeat; }
```

固定背景图像[**background-attachment**]

CSS 属性 **background-attachment** 用于指定背景图像是固定在屏幕上的、还是随着它所在的元素而滚动的。

一个固定的背景图像不会随着用户滚动页面而发生滚动（它是固定在屏幕上的），而一个非固定的背景图像会随着页面的滚动而滚动。

下表概括了 **background-attachment** 的两种不同取值。你可以点击示例察看二者的区别。

值	描述	示例
background-attachment:scroll	图像会跟随页面滚动——非固定的	显示示例
background-attachment:fixed	图像是固定在屏幕上的	显示示例

例如，下面的代码将背景图像固定在屏幕上。

```
body {
  background-image: url("butterfly.gif");
  background-repeat: no-repeat;
  background-attachment: fixed;
}
```

放置背景图像[background-position]

缺省地，背景图像将被放在屏幕的左上角。但是，你可以通过 CSS 属性 `background-position` 来修改这一缺省设置，将背景图像摆放在屏幕上你觉得满意的地方。

设置 `background-position` 属性的值有多种方式。不过，它们都是坐标的格式。举例来说，值“100px 200px”表示背景图像将被放置在位于距浏览器窗口左边 100 像素、顶部 200 像素处。

坐标可以是以百分比或固定单位（比如像素、厘米等）作为单位的值，也可以是“top”、“bottom”、“center”、“left”和“right”这些值。

下表给出了一些例子。

值	描述	示例
<code>background-position: 2cm 2cm</code>	图像被放置在页面内距左边 2 厘米、顶部 2 厘米的地方	显示示例
<code>background-position: 50% 25%</code>	图像被放置在页面内水平居中、离顶部四分之一处	显示示例
<code>background-position: top right</code>	图像被放置在页面的右上角	显示示例

在下例中，背景图像被放置在页面的右下角：

```
body {
  background-image: url("butterfly.gif");
  background-repeat: no-repeat;
  background-attachment: fixed;
  background-position: right bottom;
}
```

缩写[background]

CSS 属性 `background` 是上述所有与背景有关的属性的缩写用法。

使用 `background` 属性可以减少属性的数目，因此令样式表更简短易读。

比如说下面五行代码：

```
background-color: #FFCC66;
background-image: url("butterfly.gif");
background-repeat: no-repeat;
```

```
background-attachment: fixed;  
background-position: right bottom;
```

如果使用 **background** 属性的话，实现同样的效果只需一行代码即可搞定：

```
background: #FFCC66 url("butterfly.gif") no-repeat fixed right bottom;
```

各个值应按下列次序来写：

```
[background-color] | [background-image] | [background-repeat] | [background-attach  
ment] | [background-position]
```

如果省略某个属性不写出来，那么将自动为它取缺省值。比如，如果去掉 **background-attachment** 和 **background-position** 的话：

```
background: #FFCC66 url("butterfly.gif") no-repeat;
```

这两个未指定值的属性将被设置为缺省值：**scroll** 和 **top left**。

三：字体

CSS 字体属性主要包括下面几个：

```
font-family, font-style, font-variant, font-weight, font-size  
, font
```

字体族[font-family]

CSS 属性 **font-family** 的作用是设置一组按优先级排序的字体列表，如果该列表中的第一个字体未在访问者计算机上安装，那么就尝试列表中的下一个字体，依此类推，直到列表中的某个字体是已安装的。

有两种类型的名称可用于分类字体：字体族名称（**family-name**）和族类名称（**generic family**）。下面来解释这两个术语。

字体族名称（就是我们通常所说的“字体”）的例子包括“**Arial**”、“**Times New Roman**”、“**宋体**”、“**黑体**”等等。

一个族类是一组具有统一外观的字体族。**sans-serif** 就是一例，它代表一组没有“脚”的字体。

你在给出字体列表时，自然应把首选字体放在前面、把候选字体放在后面。建议你在列表的最后给出一个族类（**generic family**），这样，当没有一个指定字体可用时，页面至少可以采用一个相同族类的字体来显示。

下面是一个按优先级排列的字体列表的例子：

```
h1 {font-family: arial, verdana, sans-serif;}  
h2 {font-family: "Times New Roman", serif;}
```

h1 标题将采用 Arial 字体显示。如果访问者的计算机未安装 Arial, 那么就使用 Verdana 字体。假如 Verdana 字体也没安装的话, 那么将采用一个属于 sans-serif 族类的字体来显示这个 h1 标题。

注意我们为“Times New Roman”采用的写法: 因为其中包含空格, 所以我们用引号将它括起来。

字体样式[font-style]

CSS 属性 font-style 定义所选字体的显示样式: normal (正常)、italic (斜体) 或 oblique (倾斜)。在下例中, 所有 h2 标题都将显示为斜体。

```
h2 {font-family: "Times New Roman", serif; font-style: italic;}
```

字体变化[font-variant]

CSS 属性 font-variant 的值可以是: normal (正常) 或 small-caps (小体大写字母)。small-caps 字体是一种以小尺寸显示的大写字母来代替小写字母的字体。不太明白? 我们来看几个例子:

如果 font-variant 属性被设置为 small-caps, 而没有可用的支持小体大写字母的字体, 那么浏览器多半会将文字显示为正常尺寸 (而不是小尺寸) 的大写字母。

```
h1 {font-variant: small-caps;}
```

```
h2 {font-variant: normal;}
```

字体浓淡[font-weight]

CSS 属性 font-weight 指定字体显示的浓淡程度。其值可以是 normal (正常) 或 bold (加粗)。有些浏览器甚至支持采用 100 到 900 之间的数字 (以百为单位) 来衡量字体的浓淡。

```
p {font-family: arial, verdana, sans-serif;}
```

```
td {font-family: arial, verdana, sans-serif; font-weight: bold;}
```

字体大小[font-size]

字体的大小用 CSS 属性 font-size 来设置。

字体大小可通过多种不同单位 (比如像素或百分比等) 来设置。在本教程中, 我们将关注于最常用和最合适的单位。比如:

```
h1 {font-size: 30px;}
```

```
h2 {font-size: 12pt;}
```

```
h3 {font-size: 120%;}
```

```
p {font-size: 1em;}
```

上面四种单位有着本质的区别。‘px’和‘pt’将字体设置为固定大小, 而‘%’和‘em’允许页面浏览者自行调整字体的显示尺寸

缩写[font]

CSS 属性 font 是上述各有关字体的 CSS 属性的缩写用法。

比如说下面四行应用于 `p` 元素的代码：

```
p {  
  font-style: italic;  
  font-weight: bold;  
  font-size: 30px;  
  font-family: arial, sans-serif;  
}
```

如果用 `font` 属性的话，上述四行代码可简化为：

```
p {font: italic bold 30px arial, sans-serif; }
```

`font` 属性的值应按以下次序书写：

`font-style | font-variant | font-weight | font-size | font-family`

四：文本

文本主要包括下列 CSS 属性：

`text-indent`.,`text-align`,`text-decoration`,`letter-spacing`.

`text-transform`

文本缩进[`text-indent`]

CSS 属性 `text-indent` 用于为段落设置首行缩进，以令其具有美观的格式。在下例中，我们为采用 `p` 元素的段落应用了 30 像素的首行缩进。

```
p {text-indent: 30px;}
```

文本对齐[`text-align`]

CSS 属性 `text-align` 与 HTML 属性 `align` 的功能相同。该属性的值可以是：`left`（左对齐）、`right`（右对齐）或者 `center`（居中）。除了上面三种选择以外，你还可以将该属性的值设为 `justify`（两端对齐），即伸缩行中的文字以左右靠齐。报刊杂志经常采用这种布局。

在下例中，标题（`th`）中的文字被设置为右对齐，而表中数据（`td`） 被设置为居中。正常的文本段落被设置为两端对齐。

```
th {text-align: right;}  
td {text-align: center;}  
p {text-align: justify; }
```

文本装饰[`text-decoration`]

CSS 属性 `text-decoration` 令我们可以为文本增添不同的“装饰”或“效果”。例如，你可以为文本增添下划线、删除线、上划线等等。在接下来的例子中，我们为 `h1` 标题增添了下划线，为 `h2` 标题增添了上划线，为 `h3` 标题增添了删除线。

```
h1 {text-decoration: underline;}  
h2 {text-decoration: overline;}
```

```
h3 {text-decoration: line-through;}
```

字符间距[**letter-spacing**]

CSS 属性 **letter-spacing** 用于设置文本的水平字间距。我们可以把期望的字间距宽度作为这个属性的值。例如，假如你希望 **p** 元素里的文本段落的字间距为 3 个像素，而 **h1** 标题的字间距为 6 个像素，代码可以这样写：

```
h1 {letter-spacing: 6px; }
```

```
p {letter-spacing: 3px;}
```

文本转换[**text-transform**]

CSS 属性 **text-transform** 用于控制文本的大小写。无论字母本来的大小写，你可以通过该属性令它首字母大写（**capitalize**）、全部大写（**uppercase**）或者全部小写（**lowercase**）。

比如，单词“headline”在展现给网页浏览者时，可以是“HEADLINE”或者“Headline”。

text-transform 属性有四个可选值：

capitalize 将每个单词的首字母转换为大写。

uppercase 所有字母都转换为大写。

lowercase 所有字母都转换为小写

none 不作任何转换——文本怎么写的就怎么显示。

五：链接

在前面讲到的属性也可以应用到链接上（比如修改颜色、字体、添加下划线等）。但不同的是，**CSS** 允许你根据链接是未访问的、已访问的、活动的、是否有鼠标悬停等分别定义不同的属性。这样，我们便可为网站增添奇特而有用的效果。你需要通过伪类（**pseudo-class**）来控制这些效果。

伪类是什么？

伪类（**pseudo-class**）令你可以在为 **HTML** 元素定义 **CSS** 属性时将条件和事件考虑在内。

我们来看一个例子。正如你所知道的，在 **HTML** 里，链接是通过 **a** 元素来定义的。因此，在 **CSS** 里，我们可以将 **a** 作为一个选择器

```
a {color: blue;}
```

一个链接可以有不同的状态。例如，它可以是已访问过的，也可以是未访问过的。你可以通过伪类分别为访问过的链接和未访问过的链接设置不同的样式。

```
a:link {color: blue;}
```

```
a:visited {color: red;}
```

为未访问过的链接和已访问过的链接分别使用伪类 **a:link** 和 **a:visited**。活动的链接对应的伪类为 **a:active**，有鼠标悬停的链接对应的伪类为 **a:hover**。

伪类： **active**

伪类:**active** 用于活动的链接（即获得当前焦点的链接）。

```
a:active {background-color: #FFFF00;}
```

伪类: **hover** 用于有鼠标悬停的链接。

如果你要当鼠标光标移到链接上时将链接显示为橙色斜体，那么 CSS 可以这样写：

```
a:hover {color: orange;
        font-style: italic; }
```

如何去掉链接的下划线是一个常见的问题。

```
a {text-decoration:none;}
```

六：元素的分类与标识（**class** 和 **id**）

1.用 **class** 对元素进行分类

我们希望白葡萄酒的链接全部显示为黄色，红葡萄酒的链接全部显示为红色，其余的链接显示为缺省的兰色。为了实现这一要求，我们将链接分为两类。对链接的分类是通过为链接设置 HTML 属性 **class** 实现的。参加如下代码：

```
<p>制造白葡萄酒的葡萄： </p>
<ul>
<li><a href="ri.htm" class="whitewine">雷司令（Riesling） </a></li>
<li><a href="ch.htm" class="whitewine">夏敦埃（Chardonnay） </a></li>
<li><a href="pb.htm" class="whitewine">白比诺（Pinot Blanc） </a></li>
</ul>
<p>制造红葡萄酒的葡萄： </p>
<ul>
<li><a href="cs.htm" class="redwine">卡百内索维农（Cabernet Sauvignon）
</a></li>
<li><a href="me.htm" class="redwine">墨尔乐（Merlot） </a></li>
<li><a href="pn.htm" class="redwine">黑比诺（Pinot Noir） </a></li>
</ul>
```

然后，我们就可以为白葡萄酒和红葡萄酒的链接分别应用不同的风格了。

```
a {color: blue;}
a.whitewine {color: #FFBB00;}
a.redwine {color: #800000; }
```

如上例所示，你可以通过在样式表里利用 **classname** 来为属于某一类的元素定义 CSS 属性。

利用 **id** 标识元素

除了可以对元素进行分类以外，你还可以标识单个元素。这是通过 HTML 属性 **id** 实现的。HTML 属性 **id** 的特别之处在于，在同一 HTML 文档中不能有两个具有相同 **id** 值的元素。文档中的每个 **id** 值都必须是唯一的。在其他情况下，你应该使用 **class** 属性。下面，我们来看一个使用 **id** 属性的例子：

```
<h1 id="c1">第 1 章</h1>
```

...

```
<h2 id="c1-1">第 1.1 节</h2>
```

...

```
<h2 id="c1-2">第 1.2 节</h2>
```

假如我们要求第 1.2 节显示为红色，那么 CSS 可以这样写：

```
#c1-2 {  
    color: red;  
}
```

如上例所示，你可以在样式表里通过 **#id** 为特定元素定义 CSS 属性。

（十七）CSS 基础积累总结（下）

七.组织元素（**span** 和 **div**）

span 和 **div** 元素用于组织和结构化文档，并经常联合 **class** 和 **id** 属性一起使用。

在这一课中，我们将进一步探究 **span** 和 **div** 的用法，因为这两个 HTML 元素对于 CSS 是很重要的。

- 用 **span** 组织元素
- 用 **div** 组织元素

用 **span** 组织元素

span 元素可以说是一种中性元素，因为它不对文档本身添加任何东西。但是与 CSS 结合使用的话，**span** 可以对文档中的部分文本增添视觉效果。

让我们用一句本杰明·弗兰克林（Benjamin Franklin）的名言来举个例子：

<p>早睡早起

使人健康、富裕又聪颖。</p>

假设我们想用红色来强调弗兰克林先生所认为的“不要在睡眠中度过一天”的好处，我们可以用 **** 标签来标记上述每一点好处。然后，我们将这几个 **span** 设置为相同的 **class**。这样，我们稍后就可以在样式表里针对这个 **class** 定义特定的样式。

<p>早睡早起

使人健康、

富裕

和聪颖。</p>

相应的 CSS 代码如下：

```
span.benefit {  
    color:red;  
}
```

当然，你也可以采用 **id** 来为 **span** 元素添加样式。不过正如我们在上一课所学的，如果采用 **id** 的话，你必须为这三个 **span** 元素各自分别指定一个唯一的 **id**。

用 **div** 组织元素

如前面例子所示，**span** 的使用局限在一个块元素内，而 **div** 可以被用来组织一个或多个块元素。除去这点区别，**div** 和 **span** 在组织元素方面相差无几。让我们来看一个例子。我们将历届美国总统按其所属的政营分别组织为两个列表：

```
<div id="democrats">  
<ul>  
<li>富兰克林·D·罗斯福</li>  
<li>哈利·S·杜鲁门</li>
```

```

<li>约翰·F·肯尼迪</li>
<li>林登·B·约翰逊</li>
<li>吉米·卡特</li>
<li>比尔·克林顿</li>
</ul>
</div>
<div id="republicans">
<ul>
<li>德怀特·D·艾森豪威尔</li>
<li>理查德·尼克松</li>
<li>杰拉尔德·福特</li>
<li>罗纳德·里根</li>
<li>乔治·布什</li>
<li>乔治·W·布什</li>
</ul>
</div>

```

在这里，我们可以采用跟上例同样的方法来处理样式表：

```

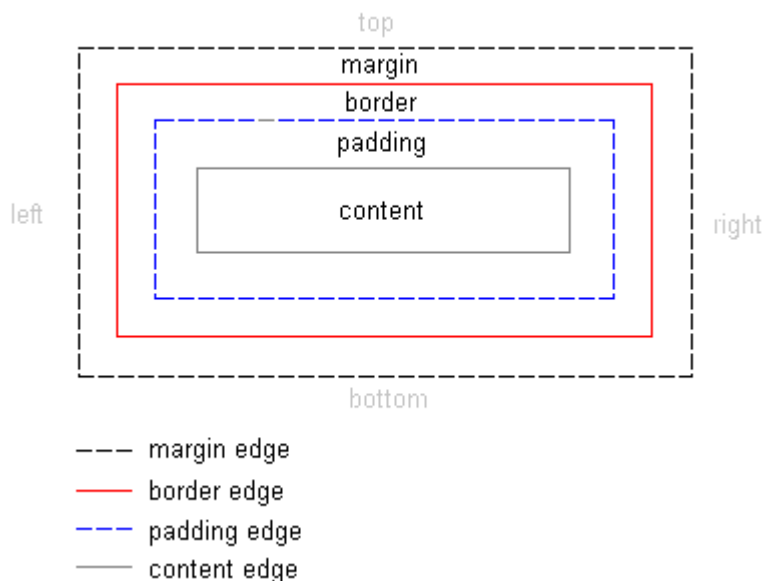
#democrats {
    background:blue;
}
#republicans {
    background:red;
}

```

八：盒状模型

CSS 中的盒状模型（box model）用于描述一个为 HTML 元素形成的矩形盒子。盒状模型还涉及为各个元素调整外边距（margin）、边框（border）、内边距（padding）和内容的具体操作。下图显示了盒状模型的结构：

CSS 中的盒状模型



上面的图示看上去可能感觉有点理论化，好吧，让我们试着用一个实例来解释盒状模型。在我们的例子中，有一个标题和一些文本。该例的 **HTML** 代码如下（摘自世界人权宣言）：

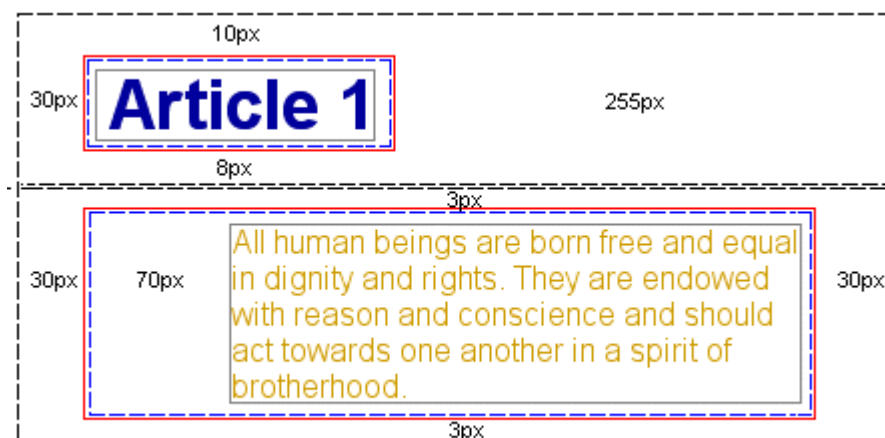
```
<h1>Article 1:</h1>
<p>All human beings are born free
and equal in dignity and rights.
They are endowed with reason and conscience
and should act towards one another in a
spirit of brotherhood</p>
```

通过添加一些颜色及字体信息，该例可以有以下显示效果：

Article 1

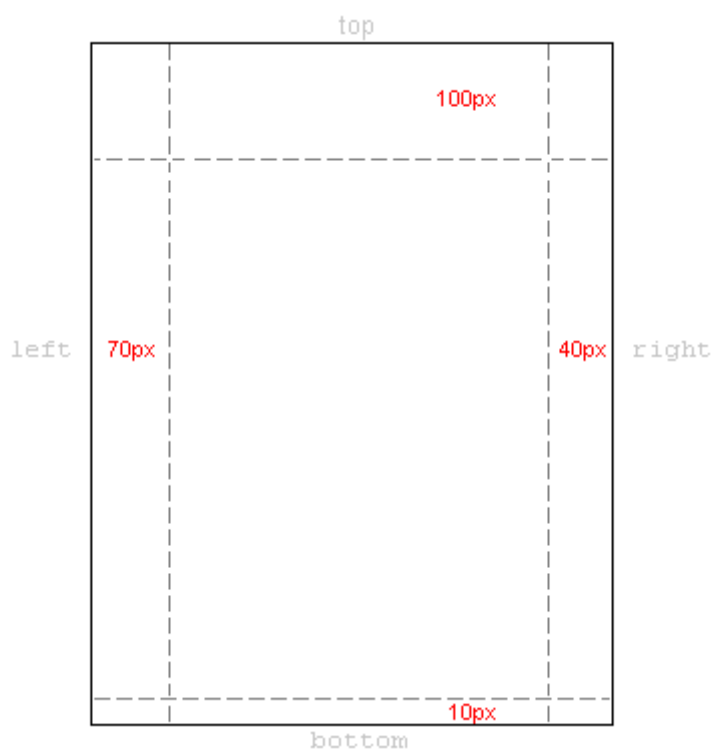
All human beings are born free and equal
in dignity and rights. They are endowed
with reason and conscience and should
act towards one another in a spirit of
brotherhood.

这个例子包含了两个元素：h1 和 p。这两个元素的盒状模型如下图所示：



为元素设置外边距

一个元素有上 (top)、下 (bottom)、左 (left)、右 (right) 四个边。外边距 (margin) 表示从一个元素的边到相邻元素 (或者文档边界) 之间的距离。可以参考第 9 课的图示。在下面这个例子中，我们将了解如何为文档本身 (即 body 元素) 定义外边距。下图显示了我们对外边距的要求。



满足上述要求的 CSS 代码如下：

```
body {
    margin-top: 100px;
    margin-right: 40px;
```

```
        margin-bottom:10px;
        margin-left:70px;
    }
```

或者你也可以采用一种较优雅的缩写形式：

```
    body {
        margin: 100px 40px 10px 70px;
    }
```

几乎所有元素都可以采用跟上面一样的方法来设置外边距。例如，我们可以为所有用<p>标记的文本段落定义外边距：

```
    body {
        margin: 100px 40px 10px 70px;
    }
    p {
        margin: 5px 50px 5px 50px;
    }
```

为元素设置内边距

内边距（padding）也可以被理解成“填充物”。这样理解是合理的，因为内边距并不影响元素间的距离，它只定义元素的内容与元素边框之间的距离。

下面我们通过一个简单的例子来说明内边距的用法。在这个例子中，所有标题都具有背景色：

```
    h1 {
        background: yellow;
    }
    h2 {
        background: orange;
    }
```

通过为标题设置内边距，你可以控制在标题文本周围填充多少空白：

```
    h1 {
        background: yellow;
        padding: 20px 20px 20px 80px;
    }
    h2 {
        background: orange;
        padding-left:120px;
    }
```

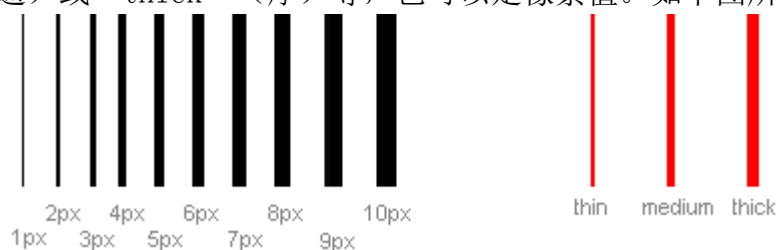
边框

边框 (border) 可以有多种用途，比如作为装饰元素或者作为划分两物的分界线。在设置边框方面，CSS 为你提供了无尽选择。

- `border-width`
- `border-color`
- `border-style`
- 一些示例
- 缩写 `[border]`

边框宽度[border-width]

边框宽度由 CSS 属性 `border-width` 定义，其值可以是 “thin” (薄)、“medium” (普通) 或 “thick” (厚) 等，也可以是像素值。如下图所示：



边框颜色[border-color]



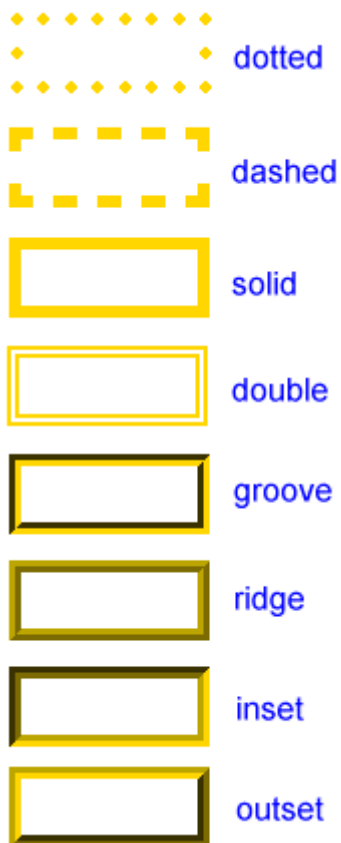
CSS 属性 `border-color` 用于定义边框的颜色。其值就是正常的颜色值，例如：

“#123456”、“`rgb(123, 123, 123)`”、“yellow” 等。

边框样式[border-style]

边框样式有多种可供选择。下图显示了 8 种不同样式的边框在 Internet Explorer 5.5 里的实际显示效果。在这个例子里，我们为这 8 种边框都选择了 “金色 (gold)” 作为边框颜色、“厚(thick)” 作为边框宽度。当然，这只是个例子，你可以为边框设置别的颜色和厚度。

如果你不想有任何边框，可以为它取值为 “none” 或者 “hidden”。



一些示例

我们可以将上面三个有关边框的 CSS 属性组合起来使用，从而制造出多种多样的变化。来举个例子，我们要为一个文档中的 h1、h2、ul 和 p 等元素分别定义不同的边框。尽管其显示效果也许并不那么美观，但是它很好地向你展示了多种变化的可能：

```
h1 {  
    border-width: thick;  
    border-style: dotted;  
    border-color: gold;  
}  
h2 {  
    border-width: 20px;  
    border-style: outset;  
    border-color: red;  
}  
p {  
    border-width: 1px;  
    border-style: dashed;
```



```
        border-color: blue;
    }
    ul {
        border-width: thin;
        border-style: solid;
        border-color: orange;
    }
```

我们也可以为上边框、下边框、右边框、左边框分别指定特定的 CSS 属性。具体做法如下例所示：

```
h1 {
    border-top-width: thick;
    border-top-style: solid;
    border-top-color: red;

    border-bottom-width: thick;
    border-bottom-style: solid;
    border-bottom-color: blue;

    border-right-width: thick;
    border-right-style: solid;
    border-right-color: green;

    border-left-width: thick;
    border-left-style: solid;
    border-left-color: orange;
}
```

缩写[border]

跟许多其他属性一样，你也可以将有关边框的 CSS 属性缩写为一个 border 属性。让我们看一个例子：

```
p {
    border-width: 1px;
    border-style: solid;
    border-color: blue;
}
```

可被缩写为：

```
p {  
    border: 1px solid blue;  
}
```

设定宽度[width]

你可以通过 width 属性来设定一个元素的宽度，即在水平方向上的尺寸。

下面是一个简单的例子，它为我们提供了一个可以容纳文本的盒子：

```
div.box {  
    width: 200px;  
    border: 1px solid black;  
    background: orange;  
}
```

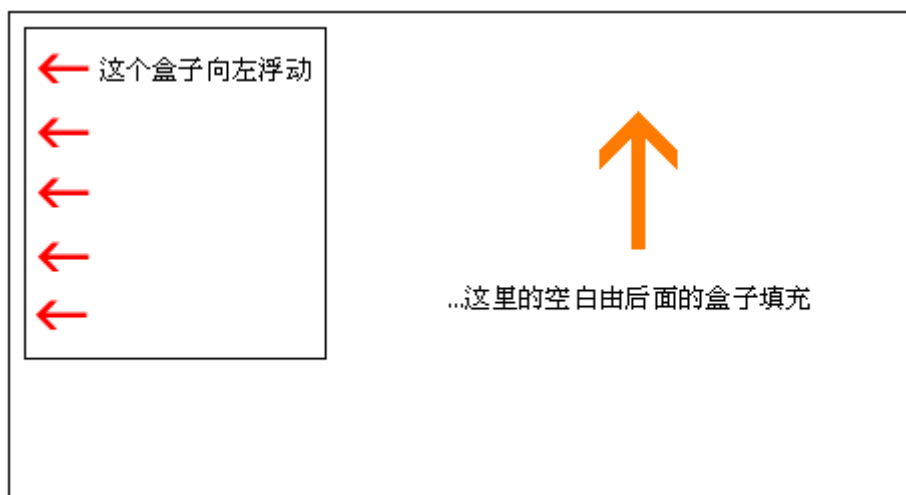
设定高度[height]

注意上一个例子，盒子里内容的长短决定了盒子的高度。你可以通过 height 属性来设定一个元素的高度。比方说，我们要把上面那个例子中的盒子高度设定为 500 像素：

```
div.box {  
    height: 500px;  
    width: 200px;  
    border: 1px solid black;  
    background: orange;  
}
```

浮动元素（float）

我们可以通过 CSS 属性 float 令元素向左或向右浮动。也就是说，令盒子及其中的内容浮动到文档（或者是上层盒子）的右边或者左边（参见第 9 课关于盒状模型的描述）。下图阐明了其原理：



如何实现这个效果？

上例的 HTML 代码如下所示：

```
<div id="picture">
    
</div>
```

```
<p>causas naturales et antecedentes,
idcirco etiam nostrarum voluntatum...</p>
```

要实现图片向左浮动、而且被文字环绕的效果，你只需先设定图片所在盒子的宽度，然后再把 CSS 属性 `float` 设为 `left` 即可：

```
#picture {
    float:left;
    width: 100px;
}
```

另一个例子：列

浮动也可以用于实现在文档中分列。要创建多个列，你需要在 HTML 代码里用 `div` 来结构化想要的各个列：

```
<div id="column1">
    <p>Haec disserens qua de re agatur
```

```

        et in quo causa consistat non videt...</p>
</div>

<div id="column2">
    <p>causas naturales et antecedentes,
    idcirco etiam nostrarum voluntatum...</p>
</div>

<div id="column3">
    <p>nam nihil esset in nostra
    potestate si res ita se haberet...</p>
</div>

```

下面，我们把各列的宽度设定为“33%”，并通过定义 float 属性令各列向左浮动：

```

#column1 {
    float:left;
    width: 33%;
}

#column2 {
    float:left;
    width: 33%;
}

#column3 {
    float:left;
    width: 33%;
}

```

float 属性的值可以是 **left**、**right** 或者 **none**。

clear 属性

CSS 属性 clear 用于控制浮动元素的后继元素的行为。

缺省地，后继元素将向上移动，以填补由于前面元素的浮动而空出的可用空间。在前面的例子中，文本自动上移到了比尔盖茨的图片旁。

clear 属性的值可以是 **left**、**right**、**both** 或 **none**。原则是这样的：如果一个盒子的 clear 属性被设为“both”，那么该盒子的上边距将始终处于前面的浮动盒子（如果存在的话）的下边距之下。

```
<div id="picture">
    
</div>

<h1>Bill Gates</h1>

<p class="floatstop">causas naturales et antecedentes,
idcirco etiam nostrarum voluntatum...</p>
```

要避免文本上移到图片旁，我们可以在 CSS 中添加以下代码：

```
#picture {
    float:left;
    width: 160px;
}

.floatstop {
    clear:both;
}
```

元素的定位

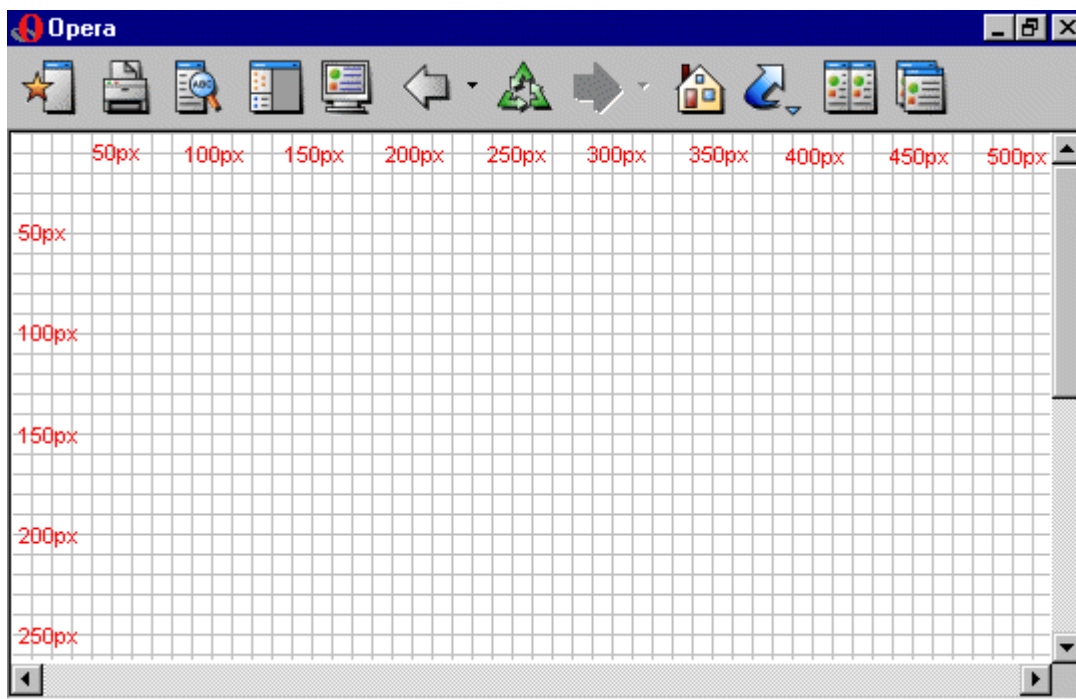
CSS 定位令你可以将一个元素精确地放在页面上你所指定的地方。联合使用定位与浮动（参见第 13 课），你将能够创建多种高级而精确的布局。

本课我们将讨论以下内容：

- CSS 定位的原理
- 绝对定位
- 相对定位

CSS 定位的原理

把浏览器窗口想象成一个坐标系统：



CSS 定位的原理是：你可以将任何盒子（**box**）放置在坐标系统的任何位置上。

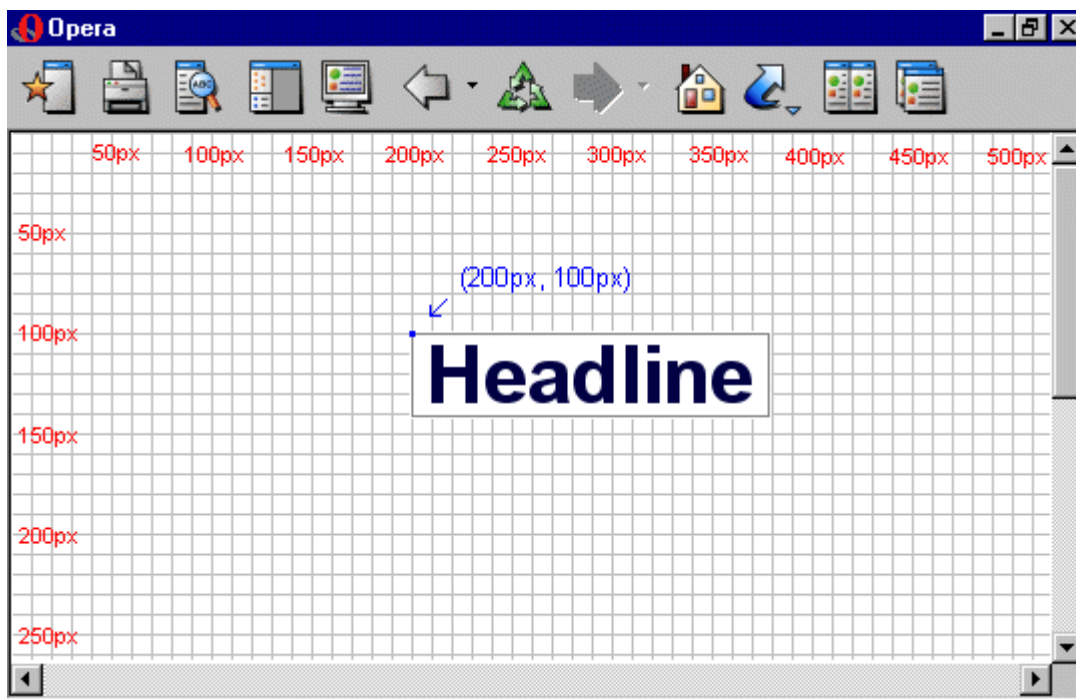
假设我们要放置一个标题。通过使用盒状模型（参见第 9 课），标题将显示如下：

Headline

如果我们要把这个标题放置在距文档顶部 100 像素、左边 200 像素的地方，我们可以在 CSS 中输入以下代码：

```
h1 {  
    position: absolute;  
    top: 100px;  
    left: 200px;  
}
```

得到的显示效果如下：



正如你所看到的，采用 CSS 定位技术来放置元素是非常精确的。相对于使用表格、透明图像或其他方法而言，CSS 定位要简单得多。

绝对定位

一个采用绝对定位的元素不获得任何空间。这意味着：该元素在被定位后不会留下空位。

要对元素进行绝对定位，应将 `position` 属性的值设为 **absolute**。接着，你可以通过属性 **left**、**right**、**top** 和 **bottom** 来设定将盒子放置在哪里。

举个绝对定位的例子，假如我们要在文档的四个角落各放置一个盒子：

```
#box1 {  
    position: absolute;  
    top: 50px;  
    left: 50px;  
}
```

```
#box2 {  
    position: absolute;  
    top: 50px;  
    right: 50px;  
}
```

```
#box3 {
```

```
        position: absolute;
        bottom: 50px;
        right: 50px;
    }

    #box4 {
        position: absolute;
        bottom: 50px;
        left: 50px;
    }
```

- [显示示例](#)

相对定位

要对元素进行相对定位，应将 `position` 属性的值设为 **relative**。绝对定位与相对定位的区别在于计算位置的方式。

采用相对定位的元素，其位置是**相对于它在文档中的原始位置**计算而来的。这意味着，相对定位是通过将元素从原来的位置向右、向左、向上或向下移动来定位的。采用相对定位的元素会获得相应的空间。

举个相对定位的例子，我们可以相对于三张图片在页面上的原始位置来对它们进行相对定位。注意这些图片将在文档中各自的原始位置处留下空位。

```
#dog1 {
    position: relative;
    left: 350px;
    bottom: 150px;
}

#dog2 {
    position: relative;
    left: 150px;
    bottom: 500px;
}

#dog3 {
```



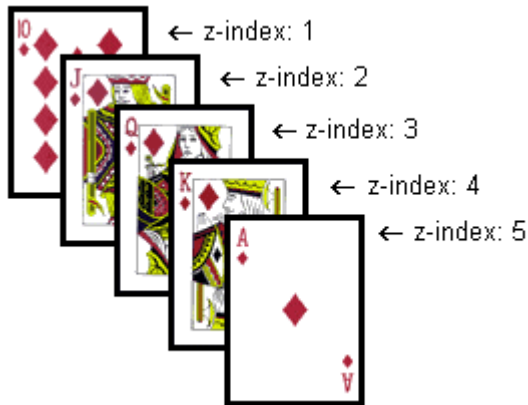
```
position: relative;
left: 50px;
bottom: 700px;
}
```

用 **z-index** 进行层次堆叠

CSS 可以处理高度、宽度、深度三个维度。在前面的课程中，我们已经了解了前两个维度。在本课中，我们将学习如何令不同元素具有层次。简言之，就是关于元素堆叠的次序问题。

为此，你可以为每个元素指定一个数字（z-index）。其原理是：数字较大的元素将叠加在数字较小的元素之上。

比方说，我们正在打扑克，并且拿了一手同花大顺。我们可以通过为各张牌设定一个 z-index 的方式来表示这手牌：



在这个例子中，我们采用了 1-5 五个连续的数字来表示堆叠次序，但是你也可以用五个不同的其他数字来取得同样的效果。这里的要点在于：用数字的大小次序反映希望的堆叠次序。

扑克牌这个例子的代码可以这样写：

```
#ten_of_diamonds {
    position: absolute;
    left: 100px;
    top: 100px;
    z-index: 1;
}
```

```
#jack_of_diamonds {
    position: absolute;
    left: 115px;
```

```
        top: 115px;
        z-index: 2;
    }

    #queen_of_diamonds {
        position: absolute;
        left: 130px;
        top: 130px;
        z-index: 3;
    }

    #king_of_diamonds {
        position: absolute;
        left: 145px;
        top: 145px;
        z-index: 4;
    }

    #ace_of_diamonds {
        position: absolute;
        left: 160px;
        top: 160px;
        z-index: 5;
    }
```

(十八) JSP 基本语法与动作指令

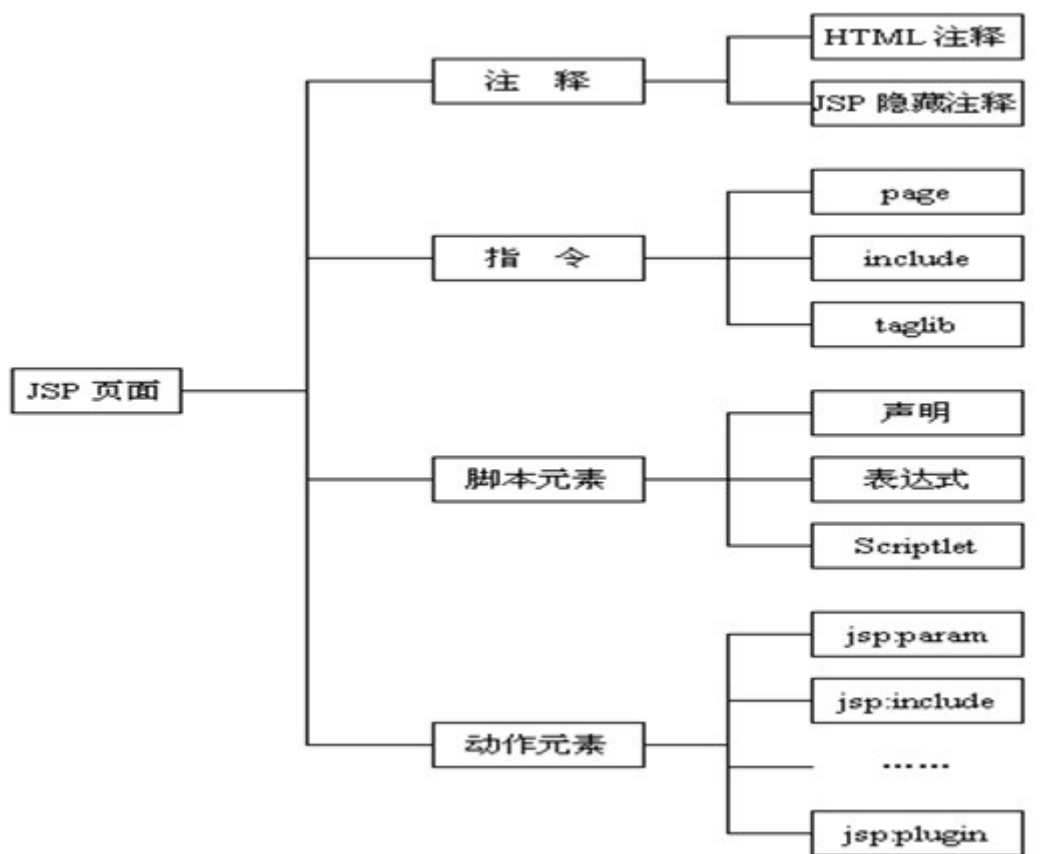
本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

Jsp,通常的被大家认为是做网页的前台界面，我刚学习的时候，说实话，真没把他当回事，学的也是囫圇吞枣，有时用到一些指令都需要现查，所以，基础知识的掌握是还是很有必要的，先总体说一下吧：

JSP 就是把 Java 代码嵌套在 HTML 中，所以 JSP 程序的结构可以分为两大部分：一部分是静态的 HTML 代码；另一部分是动态的 Java 代码和 JSP 自身的标签和指令；当 JSP 页面第一次被请求的时候，服务器的 JSP 编译器会把 JSP 页面编译成对应的 Java 代码，根据动态 Java 代码执行的结果，生成对应的纯 HTML 的字符串返回给浏览器，这样就可以把动态程序的结果展示给用户。

JSP 页面的构成：



一：JSP 页面中包含三种注释

HTML 格式注释（客户端注释）主要是用于在客户端动态地显示一个注释，格式如下：<!--注释内容[<%=expression%>] -->可通过查看 html 源代码看到。

JSP 代码注释（服务器端注释）也叫 JSP 隐藏注释，在 JSP 源代码中，它不会被 JSP 引擎处理，也不会客户端的 Web 浏览器上显示，格式如下：

<%--注释内容 --%>

Java 语言注释和 Java 中的注释一样不过需写在<%%>内。有单行注释，多行注释。例如<% //单行注释内容 %>、<% /* 多行注释内容 */ %>

二：指令

在 JSP 中，指令主要用来与 JSP 引擎进行沟通，并为 JSP 页面设置全局变量、声明类以及 JSP 要实现的方法和输出内容的类型等。需要注意的是，指令元素在 JSP 整个页面范围内有效，并且它不在客户端产生任何输出。使用指令的格式如下：<%@ 指令名 属性 1="值 1" 属性 2="值 2" ... %>

JSP 包括三种指令：page 指令、include 指令和 taglib 指令。

page 指令：定义与 JSP 页面相关的属性，并和 JSP 引擎进行通信。一个 JSP 页面可以包含多个 page 指令，指令之间是相互独立的，并且指令中除 import 属性之外的每个属性只能定义一次，否则在 JSP 页面的编译过程中将出现错误。

page 指令可以运用于整个 JSP 文件，一般来说，page 指令可以放在 JSP 页面的任何位置，但为了便于程序的阅读和格式规范，通常将 page 指令放在 JSP 页面的开始部分。

例如：

```
<%@ page language="java" import="java.util.*" pageEncoding="utf-8"%>
```

include 指令：用来指定 JSP 文件被编译时需要插入的资源，这个资源可以是文本、代码、HTML 文件或 JSP 文件。

格式为：<%@include file="relativeURL"%>

其中，relativeURL 表示要包含的文件路径。如果路径以“/”开头，则表示该路径是参照 JSP 应用的上下关系路径，如果路径直接以目录名或文件名开头，则表示该路径是正在使用的 JSP 文件的当前路径。一旦 JSP 文件完成编译，该资源内容就不可变，要改变该资源内容，必须重新编译 JSP 文件。

利用 `include` 指令，可以将一个复杂的 JSP 页面分为若干个部分，这样可以方便管理 JSP 页面。一个 JSP 页面一般可以分为三段：`head`（页头）、`body`（页体）和 `tail`（页尾）。可以将一个 JSP 页面分为 3 个不同的 JSP 页面：`head.jsp`、`body.jsp` 和 `tail.jsp`，其中 `head.jsp` 表示页头，`body.jsp` 表示页体，`tail.jsp` 表示页尾，这样对于同一网站的不同 JSP 页面，可以直接利用 `include` 指令调用 `head.jsp` 和 `tail.jsp`，仅 `body.jsp` 不同

taglib 指令：是页面使用者用来自定义标签。可以把一些需要重复显示的内容自定义成为一个标签，以增加代码的重用程度，并使页面易于维护，我们暂且不用。

三：脚本元素

脚本元素是 JSP 代码中使用最频繁的元素，它是用 Java 写的脚本代码。所有的脚本元素均是以“`<%`”标记开始，以“`%>`”标记结束，它可以分为如下三类：

声明、表达式、Scriptlet

声明：在 JSP 中，声明是用来定义在程序中使用的实体，它是一段 Java 代码，可以声明变量也可以声明方法，它以“`<%!`”标记开始，以“`%>`”标记结束，格式如下：`<%! 具体声明代码 %>`

每个声明仅在一个 JSP 页面内有效，如果要想在每个页面中都包含某些声明，可将这些声明包含在一个 JSP 页面中，然后利用前面介绍的 `include` 指令将该页面包含在每个 JSP 页面中。

表达式：以“<%=”标记开始，以“%>”标记结尾，中间的内容为 Java 一个合法的表达式，格式如下：<%=expression%> 其中 expression 为 Java 表达式。表达式在执行时会被自动转换为字符串，然后显示在 JSP 页面中。

Scriptlet: 是以“<%”标记开始，以“%>”标记结尾的一段 Java 代码，它可以包含任意合乎 Java 语法标准的 Java 代码，格式如下：

<% Java 代码 %>

四：动作指令

大多数的 JSP 处理都是通过 JSP 中的动作元素来完成的，动作元素主要是在请求处理阶段起作用，它能影响输出流和对象的创建、使用、修改等。JSP 动作元素是利用 XML（可扩展标记语言）语法写成的，它们均以“jsp”作为前缀，下面介绍几种常用的动作元素：

<jsp:include>: 允许在 JSP 页面中包含静态和动态页面。如果包含的是静态页面，则只是将静态页面的内容加入至 JSP 页面中，如果包含的是动态页面，则所包含的页面将会被 JSP 服务器编译执行。格式如下：

<jsp:include page="relativeURL"<%=expression%>" flush="true|false"/>

page: 表示所要包含的文件的相对 URL，它可以是一个字符串，也可以是一个 JSP 表达式。**flush:** 默认值为 false，若该值为 true 则表示当缓冲区满时，缓冲区将被清空。

以下是对 include 两种用法的区别，重要有两个方面的不同：

执行时间上：

静态包含：<%@ include file=" relativeURI" %> 是在翻译阶段执行

动态包含: `<jsp:include page="relativeURI" flush="true" />` 在请求处理阶段执行.

所谓动态包含是指在请求包含页面的时候遇到动态包含指令将请求转到被包含页面, 这时去编译被包含页面。静态包含是在请求包含页面时去编译包含页面, 编译时遇到静态页面包含伪码将被包含页面的内容复制到被包含页面中进行编译。

引入内容的不同:

`<%@ include file="relativeURI" %>` 引入静态文本(html,jsp), 在 JSP 页面被转化成 servlet 之前和它融和到一起.

`<jsp:include page="relativeURI" flush="true" />` 引入执行页面或 servlet 所天生的应答文本.

另外在两种用法中 file 和 page 属性都被说明为一个相对的 URL. 如果它以斜杠开头, 那么它就是一个环境相关的路径. 如果它不是以斜杠开头, 那么就是页面相关的路径, 就根据引入这个文档的页面所在的路径进行说明。

<jsp:forward>: `<jsp:forward>` 操作允许将当前的请求运行转发至另外一个静态的文件、JSP 页面或含有与当前页面相同内容的 Servlet。格式如下:

`<jsp:forward page="relativeURL|<%=expression%>" />`

注意: forward 动作指令和 HTML 中的 `<a>` 超链接标签是不同的, 在 `<a>` 中只有单击链接才能实现页面跳转, 在 forward 动作指令中一切都是可以用 Java 的代码进行控制, 可以在程序中直接决定页面跳转的方向和时机。在 forward 跳转并且传递参数的过程中, 浏览器地址栏中的地址始终是

不变的，传递的参数也不会浏览器的地址栏中显示，这也是 forward 动作指令与 HTML 中<a>超链接的另一个区别。

<jsp:useBean>标签表示用来在 JSP 页面中创建一个 BEAN 实例并指定它的名字以及作用范围。

语法：

```
<jsp:useBean id="name" scope="page | request | session | application" typeSpec />
```

其中 typeSpec 有以下几种可能的情况：

```
class="className" | class="className" type="typeName" | beanName="beanName" type="typeName" | type="typeName" |
```

注：

你必须使用 class 或 type，而不能同时使用 class 和 beanName。beanName 表示 Bean 的名字，其形式为“a.b.c”。

GetProperty 指令

<jsp:getProperty>标签表示获取 BEAN 的属性的值并将之转化为一个字符串，然后将其插入到输出的页面中。

语法：

```
<jsp:getProperty name="name" property="propertyName" />
```

注：

- 1、在使用<jsp:getProperty>之前，必须用<jsp:useBean>来创建它。
- 2、不能使用<jsp:getProperty>来检索一个已经被索引了的属性。
- 3、能够和 JavaBeans 组件一起使用<jsp:getProperty>，但是不能与 Enterprise Java Bean 一起使用。

SetProperty 指令

<jsp:setProperty>标签表示用来设置 **Bean** 中的属性值。

语法：

```
<jsp:setProperty name="beanName" prop_expr />
```

其中 prop_expr 有以下几种可能的情形：

```
property="*" | property="propertyName" | property="propertyName" param  
="parameterName" | property="propertyName" value="propertyValue"
```

param 指令

param 指令用于设置参数值，这个指令本身不能单独使用，单独的 *param*

没有实际意义，*param* 指令可与一下三个指令结合使用：

jsp:include

jsp:forward

jsp:plugin

param 指令的语法格式如下：

```
<jsp:param name="paramName" value="paramValue">
```

(十九)EL 表达式和 JSTL

一：EL 表达式：

1.定义：为了计算和输出存储在标志位置的 Java 对象的值，JSP2.0 引入了一种简洁的语言。

2.基本格式：`${表达式}`

所有的 EL 都是以 “`${`” 开始，以 “`}`” 结尾

表达式与开始符和终结符的空格被忽略

表达式的值为 null，则在页面中显示为一个空字符串，而不是 null

3.EL 表达式运算符

= 或 eq	相等 (equals)
!= 或 ne	不相等 (not equals)
< 或 lt	小于 (less than)
> 或 gt	大于 (greater than)
<= 或 le	小于等于 (less than or equals)
>= 或 ge	大于等于 (greater than or equals)

符号	说明	符号	说明
&& 或 and	逻辑与	empty	是否为null或空字符串
 或 or	逻辑或		
! 或 not	取反	?:	三元运算符

4.EL 的作用域

使用 EL 的时候，默认会以一定顺序搜索四个作用域，将最先找到的变量值显示出来。



EL中的作用域	对应关系
pageContext	当前页的pageContext对象
pageScope	把page作用域中的数据映射为一个map对象
requestScope	把request作用域中的数据映射为一个map对象
sessionScope	把session作用域中的数据映射为一个map对象
applicationScope	把application作用域中的数据映射为一个map对象
param	对应request.getParameter()
paramValues	对应request.getParameterValues()
header	对应request.getHeader()
headerValues	对应request.getHeaderValues()
cookie	对应request.getCookies()
initParam	对应ServletContext.getInitParamter()

5.EL 表达式的隐式对象

EL 提供了四个与范围有关的隐式对象，对应四个存取范围

pageScope: 范围和 JSP 的 page 相同，只限于当前页面

requestScope: 范围和 JSP 的 request 相同，范围限于一次请求

sessionScope: 范围和 JSP 的 session 相同，范围为一次会话

applicationScope: 从服务器一开始执行服务，到服务器关闭为止

在 EL 中，四个隐含对象只能单纯用来取得对应范围内的属性值

6.使用 EL 表达式的好处

1) 代替复杂代码，省去条件判断

2) 简单访问 Bean 的属性：\${user.name}

3) 使用 EL 表达式可以输出 MVC 中的内容，代码简单

例如：Servlet 的 doPost()或 doGet()方法中，保存在作用域范围内数据，可以在其它的 JSP 页面获取。如 Servlet 中有如下代码：

```
request.getSession().setAttribute(“loginedUser”,user);
```

在 JSP 页面中可以这样获取：\${loginedUser.name}

7.EL 显示 Form 表单请求参数信息

EL 提供了两个与输入有关的隐含对象：param 和 paramValues

用于获取<form></form>表单提交的信息，用来解析 request 中的参数

格式: `${param.参数名}`或`${paramValues.参数名}`

等同于 Java 中的 `request.getParameter(“参数名”)`或 `request.getParameterValues(“参数名”)`

8.总结 EL 表达式的主要功能

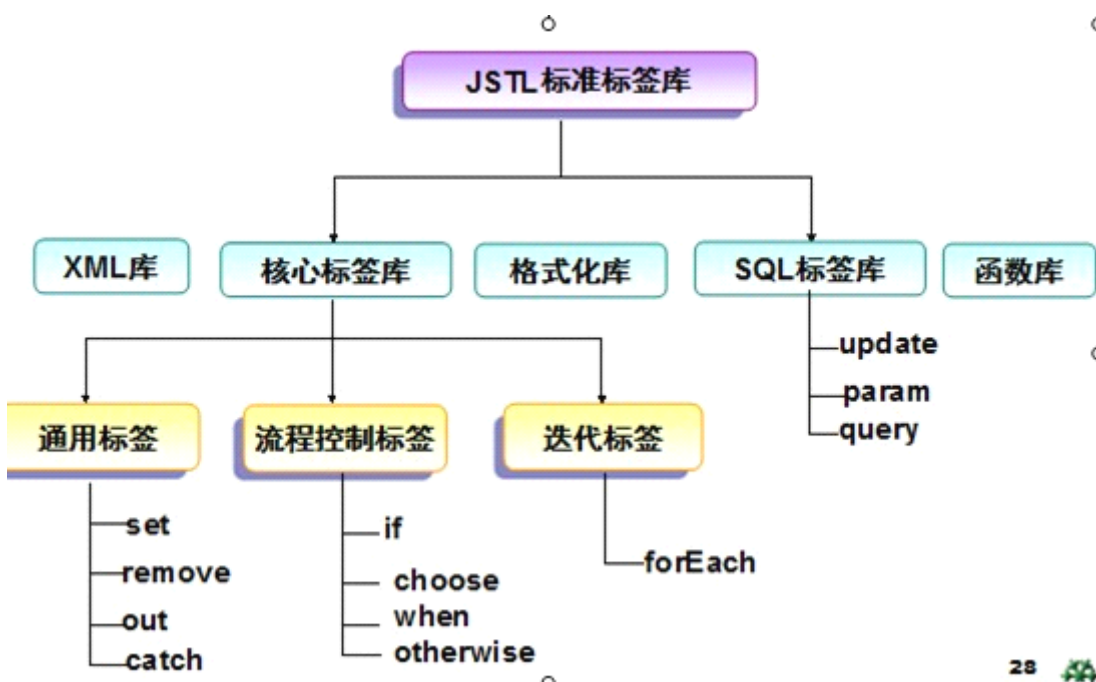
- 1) EL 的功能
- 2) 与`<jsp:getProperty />`类似
- 3) 简化`<jsp:getProperty />`
- 4) 精确的访问存储对象
- 5) Bean 属性的简略记法
- 6) 空值取代错误消息

二. JSTL

1.什么是 JSTL

JSTL (JavaServerPages Standard Tag Library) JSP 标准标签库

2.JSTL 标准标签库内的标签



3. 核心标签库

JSTL 的核心标签库标签共 13 个，从功能上可以分为 4 类：表达式控制标签、流程控制标签、循环标签、URL 操作标签。使用这些标签能够完成 JSP 页面的基本功能，减少编码工作。

- (1) 表达式控制标签：out 标签、set 标签、remove 标签、catch 标签。
- (2) 流程控制标签：if 标签、choose 标签、when 标签、otherwise 标签。
- (3) 循环标签：forEach 标签、forTokens 标签。
- (4) URL 操作标签：import 标签、url 标签、redirect 标签。

在 JSP 页面引入核心标签库的代码为：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

下面将按照功能分类，分别讲解每个标签的功能和使用方式。

1) 表达式控制标签

表达式控制分类中包括<c:out>、<c:set>、<c:remove>、<c:catch>4 个标签，现在分别介绍它们的功能和语法。

1. <c:out>标签

【功能】：用来显示数据对象（字符串、表达式）的内容或结果。

在使用 Java 脚本输出时常使用的方式为：

```
<% out.println("字符串")%>
```

```
<%=表达式%>
```

使用<c:out>标签就可以实现以上功能。

```
<c:out value="字符串">
```

```
<c:out value="EL 表达式">
```

提示：JSTL 的使用是和 EL 表达式分不开的，EL 表达式虽然可以直接将结果返回给页面，但有时得到的结果为空，<c:out>有特定的结果处理功能，EL 的单独使用会降低程序的易读性，建议把 EL 的结果输入放入<c:out>标签中。

<c:out>标签的使用有两种语法格式。

【语法 1】：

```
<c:out value="要显示的数据对象" [escapeXml="true|false"] [default="默认值"]>
```

【语法 2】：

```
<c:out value="要显示的数据对象" [escapeXml="true|false"]>默认值
```

2. <c:set>标签

功能：主要用于将变量存取于 JSP 范围中或 JavaBean 属性中。

<c:set>标签的编写共有 4 种语法格式。

语法 1：存值，把一个值放在指定（page、session 等）的 map 中。

```
<c:set value="值
```

```
1" var="name1" [scope="page|request|session|application"]>
```

含义：把一个变量名为 name1 值为“值 1”的变量存储在指定的 scope 范围内。

含义：把一个变量名为 name2，值为值 2 的变量存储在指定的 scope 范围内。

语法 3：<c:set value="值 3" target="JavaBean 对象" property="属性名"/>

含义：把一个值为“值 3”赋值给指定的 JavaBean 的属性名。相当与 setter()方法。

提示：从功能上分语法 1 和语法 2（未写）、语法 3 和语法 4（未写）的效果是一样的只是把 value 值放置的位置不同至于使用那个根据个人的喜爱，语法 1 和语法 2 是向 scope 范围内存储一个值，语法 3 和语法 4 是给指定的 JavaBean 赋值。

3. <c:remove>标签

<c:remove>标签主要用来从指定的 JSP 范围内移除指定的变量。

【语法】：

```
<c:remove var="变量名" [scope="page|request|session|application"]/>
```

其中 var 属性是必须的，scope 可以以省略。

4. <c:catch>标签：用来处理 JSP 页面中产生的异常，并将异常信息存储。

【语法】：<c:catch var="name1">

容易产生异常的代码

```
</c:catch>
```

【参数说明】：

var 表示由用户定义存取异常信息的变量的名称。省略后也可以实现异常的捕获，当就不能显示的输出异常信息。

4.流程控制标签

流程控制标签主要用于对页面简单业务逻辑进行控制。流程控制标签包含有 4 个：

<c:if>标签、<c:choose>标签、<c:when>标签和<c:otherwise>标签。下面将介绍这些标签的功能和使用方式。

1) . <c:if>标签

<c:if>同程序中的 if 作用相同，用来实现条件控制。

【语法 1】：

```
<c:if test="条件
```

```
1" var="name" [scope="page|request|session|application"]>
```

【语法 2】：

```
<c:if test="条件
```

```
2" var="name"[scope="page|request|session|application"]>
```

【参数说明】：

(1) test 属性用于存放判断的条件，一般使用 EL 表达式来编写。

(2) var 指定名称用来存放判断的结果类型为 true 或 false。

(3) scope 用来存放 var 属性存放的范围。

【使用场景】：在开发中经常会出现不同用户的权限，首先对用户名进行判断（包括进行数据库验证，该功能可以由 JavaBean 实现，使用 EL 表达式得到一个布尔型的结果），把判断的结果存放在不同的 JSP 范围内（比如常用的 session 内），这样在每个页面都可以得到该用户的权限信息，根据不同权限的用户显示不同的结果。

2) . <c:choose>、<c:when>和<c:otherwise>标签

这 3 个标签通常情况下是一起使用的,<c:choose>标签作为<c:when>和<c:otherwise>标签的父标签来使用。

【语法 1】：

[html] [view plaincopyprint?](#)

1. <c:choose>
2. <c:when>
3.//业务逻辑 1
4. </c:when>
5. <c:otherwise>
6.//业务逻辑 2
7. </c:otherwise>
8.//业务逻辑 3
9. </c:choose>
10. 【语法 2】：
11. <c:when text="条件">
12. 表达式
13. </c:when>
14. 【语法 3】：
15. <c:otherwise>
16. 表达式
17. </c:otherwise>

【参数说明】：

(1) 语法 1 为 3 个标签的嵌套使用方式,<c:choose>标签只能和<c:when>标签共同使用。

(2) 语法 2 为<c:when>标签的使用方式,该标签都条件进行判断,一般情况下和<c:choose>共同使用。

(3) <c:otherwise>不含有参数,只能跟<c:when>共同使用,并且在嵌套中只允许出现一次。

5.循环标签

循环标签主要实现迭代操作。主要包含两个标签:<c:forEach>和<c:forTokens>标签,我们主要是看<c:forEach>标签

该标签根据循环条件遍历集合(Collection)中的元素。

【语法】：

```
<c:forEach var="name" items="Collection" varStatus="StatusName" begin="begin" end="end" step="step">
```

本体内容

```
</c:forEach>
```

【参数解析】：

- (1) **var** 设定变量名用于存储从集合中取出元素。
- (2) **items** 指定要遍历的集合。
- (3) **varStatus** 设定变量名，该变量用于存放集合中元素的信息。
- (4) **begin**、**end** 用于指定遍历的起始位置和终止位置（可选）。
- (5) **step** 指定循环的步长。

其中 **varStatus** 有 4 个状态属性（见表 9-2）。

表 9-2 **varStatus** 的 4 个状态

属性名	类型	说明
index	int	当前循环的索引值
count	int	循环的次数
frist	boolean	是否为第一个位置
last	boolean	是否为第二个位置

6.SQL 标签库

JSTL 提供了与数据库相关操作的标签，可以直接从页面上实现数据库操作的功能，在开发小型网站是可以很方便的实现数据的读取和操作。**SQL** 标签库从功能上可以划分为两类：设置数据源标签、**SQL** 指令标签。

引入 **SQL** 标签库的指令代码为：

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

1) 设置数据源

使用 `<sql:setDataSource>` 标签可以实现对数据源的配置。

【语法 1】：直接使用已经存在的数据源。

```
<sql:setDataSource dataSource="dataSource"[var="name"] [scope="page|request|session|application"]/>
```

【语法 2】：使用 **JDBC** 方式建立数据库连接。

[html] [view plaincopyprint?](#)

1. `<sql:setDataSource driver="driverClass" url="jdbcURL" user="username" password="pwd">`

[html] view plaincopyprint?

1. **[var="name"]**

[html] view plaincopyprint?

1. **[scope="page|request|session|application"]>**

表 9-15 <sql:DataSource>特殊标签属性说明

参数名	说明	EL 类型	必须	默认值
dataSource	数据源	是 String Javax.sql.DataSource	否	无
var	指定存储数据源的变量名	否 String	否	无

提示：是否必须是相对的，比如说如果使用数据源则，driver、url 等就不再被使用。
如果使用 JDBC 则要用到 driver、url、user、password 属性。

提示：可以把数据连接的配置存入 session 中，如果再用到数据库连接只须配置使用 DataSource 属性。

2) SQL 操作标签

JSTL 提供了<sql:query>、<sql:update>、<sql:param>、<sql:dateParam>和<sql:transaction>这 5 个标签，通过使用 SQL 语言操作数据库，实现增加、删除、修改等操作。下面将介绍这 5 个标签的功能和使用方式。

1. <sql:query>标签用来查询数据。

【语法】：

[html] view plaincopyprint?

1. **<sql:query sql="sqlQuery" var="name" [scope="page|request|session|application"]**
- 2.
3. **[dataSource="dateSource"]**
- 4.
5. **[maxRow="maxRow"]**
- 6.
7. **[startRow="starRow"]>**

【属性说明】：见表 9-16。

表 9-16 <sql:query>标签属性说明

参数名	说明	EL 类型	必须	默认值
sql	查询数据的 SQL 语句	是 String	是	无
dataSource	数据源对象	是 String Javax.sql.DataSource	否	无
maxRow	设定最多可以暂存数据的行数	是 String	否	无
startRow	设定从那一行数据开始	是 String	否	无
var	指定存储查询结果的变量名	否 String	是	无
scope	指定结果的作用域	否 String	否	page

使用<sql:query>必须指定数据源，dataSource 是可选的，如果未给定该属性标签会在 page 范围内查找是否设置过数据源，如果没有找到将抛出异常。

一般情况下使用<sql:setDataSource>标签设置一个数据源存储在 session 范围中，当需要数据库连接时使用 dataSource 属性并实现数据库的操作。

<sql:query>的 var 属性是必须的用来存放结果集，如果没有指定 scope 范围则默认为 page，即在当前页面我们可以随时输出查询结果。结果集有一系列的属性如表 9-17 所示。

maxRows 和 startRow 属性用来操作结果集，使用 SQL 语句首先吧数据放入内存中，检查是否设置了 startRow 属性，如果设置了就从 starRow 指定的那一行开始取 maxRows 个值，如果没有设定则从第一行开始取。

表 9-17 结果集参数说明

属性名	类型	说明
rowCount	int	结果集中的记录总数
Rows	Java.util.Map	以字段为索引查询的结果
rowsByIndex	Object[]	以数字为作索引的查询结果
columnNames	String[]	字段名称数组
limitedByMaxRows	boolean	是否设置了 maxRows 属性来限制查询记录的数量

提示：limitedByMaxRows 用来判断程序是否收到 maxRows 属性的限制。并不是说设定了 maxRows 属性，得到结果集的 limitedByMaxRows 的属性都为 true，当取出的结果集小于 maxRows 时，则 maxRows 没有对结果集起到作用此时也为 false。例如可以使用 startRow 属性限制结果集的数据量。

结果集的作用就是定义了数据在页面中的显示方式。下面给出了结果集每个属性的作用。

q rowCount 属性。该属性统计结果集中有效记录的量，可以使用于大批量数据分页显示。

q Rows 属性。等到每个字段对应的值。返回的结果为：字段名={字段值…}

q rowsByIndex 属性。常用得到数据库中数据的方式，从有效行的第一个元素开始遍历，到最后一个有效行的最后一个元素。

q columnNames 属性。用于得到数据库中的字段名。

q limitedByMaxRows 属性。用于判断是否受到了 maxRows 的限制。

2) . <sql:update>标签

<sql:update>用来实现操作数据库如：使用 create、update、delete 和 insert 等 SQL 语句，并返回影响记录的条数。

【语法】：SQL 语句放在标签属性中。

```
<sql:update sql="SQL 语句" [var="name"] [scope="page|request|session|application"]  
[dataSource="dataSource"]/>
```

提示：<sql:update>标签的属性同<sql:query>标签的属性相比只减少了 maxRows 和 startRow2 个属性。其他参数用法一样。

使用<sql:update>可以实现数据表的创建、插入数据、更新数据、删除数据。使用时只须在标签中放入正确的 SQL 语句即可，同时要捕获可能产生的异常。本节只对一个简单的插入操作进行说明。

3. <sql:param>标签

<sql:param>标签用于动态的为 SQL 语句设定参数，同<sql:query>标签共同使用。可以防止 SQL 注入作用类似于 java.sql.PreparedStatement。

【语法】：

```
<sql:param value="value"/>
```

【参数说明】：

value 的作用为 SQL 中的参数赋值。

4. <sql:dateParam>标签主要用于为 SQL 标签填充日期类型的参数值。

【语法】：<sql:dateParam value="date"[type="timestamp|time|date"]/>

【参数说明】：

q value 属性：java.util.Date 类型的参数。

q type 属性：指定填充日期的类型 timestamp（全部日期和时间）、time（填充的参数为时间）、date（填充的参数为日期）。

5. <sql:transaction>标签

<sql:transaction>标签提供了数据操作的一种安全机制（即事务回滚），当操作数据库的某条 SQL 语句发生异常时，取消<sql:transaction>标签体中的所有操作，恢复原来的状态，重新对数据库进行操作。

【语法】：

[html] [view plaincopyprint?](#)

1. <sql:transaction [dataSource="dataSource"]
- 2.
3. [isolation="read_committed|read_uncommitted|repeatable|serializable"]
- 4.
5. >
- 6.
7. <sql:query>
- 8.
9. <sql:uptade>
- 10.
11. </sql:transation>

(二十) jsp 自定义标签

一、基本概念

1、标签(Tag)

标签是一种 XML 元素，通过标签可以使 JSP 网页变得简洁并且易于维护，还可以方便地实现同一个 JSP 文件支持多种语言版本。由于标签是 XML 元素，所以它的名称和属性都是大小写敏感的。

2、标签库(Tag library)

由一系列功能相似、逻辑上互相联系的标签构成的集合称为标签库。

3、标签库描述文件(Tag Library Descriptor)

标签库描述文件是一个 XML 文件，这个文件提供了标签库中类和 JSP 中对标签引用的映射关系。它是一个配置文件，和 web.xml 是类似的。

4、标签处理类(Tag Handle Class)

标签处理类是一个 Java 类，这个类继承了 TagSupport 或者扩展了 SimpleTag 接口，通过这个类可以实现自定义 JSP 标签的具体功能。

二、自定义 JSP 标签的格式

1、为了使到 JSP 容器能够使用标签库中的自定义行为，必须满足以下两个条件：

```
<% @ taglib prefix="someprefix" uri="/sometaglib" %>
```

1)从一个指定的标签库中识别出代表这种自定义行为的标签；

2)找到实现这些自定义行为的具体类。

第一个必需条件—找出一个自定义行为属于那个标签库—是由标签指令的前缀(Taglib Directive's Prefix)属性完成，所以在同一个页面中使用相同前缀的元

素都属于这个标签库。每个标签库都定义了一个默认的前缀，用在标签库的文档中或者页面中插入自定义标签。所以，你可以使用除了诸如 `jsp,jsp,x,java,servlet,sun,sunw`(它们都是在 JSP 白皮书中指定的保留字)之类的前缀。

`uri` 属性满足了以上的第二个要求。为每个自定义行为找到对应的类。这个 `uri` 包含了一个字符串，容器用它来定位 TLD 文件。在 TLD 文件中可以找到标签库中所有标签处理类的名称。

2、当 web 应用程序启动时，容器从 WEB-INF 文件夹的目录结构的 META-INF 搜索所有以 `.tld` 结尾的文件。也就是说它们会定位所有的 TLD 文件。对于每个 TLD 文件，容器会先获取标签库的 URI，然后为每个 TLD 文件和对应的 URI 创建映射关系。

在 JSP 页面中，我们仅需通过使用带有 URI 属性值的标签库指令来和具体的标签库匹配。

三、自定义 JSP 标签的处理过程

1、在 JSP 中引入标签库

```
<% @ taglib prefix="taglibprefix" uri="tagliburi" %>
```

2、在 JSP 中使用标签库标签

3、Web 容器根据第二个步骤中的 `prefix`，获得第一个步骤中声明的 `taglib` 的 `uri` 属性值

4、Web 容器根据 `uri` 属性在 `web.xml` 找到对应的元素

5. 从元素中获得对应的元素的值

6. Web 容器根据元素的值从 WEB-INF/目录下找到对应的 `.tld` 文件

7. 从.tld 文件中找到与 tagname 对应的元素
8. 从元素中获得对应的元素的值
9. Web 容器根据元素的值创建相应的 tag handle class 的实例
10. Web 容器调用这个实例的 doStartTag/doEndTag 方法完成相应的处理。

四、创建和使用一个 **Tag Library** 的基本步骤

- 1、创建标签的处理类(Tag Handler Class)
- 2、创建标签库描述文件(Tag Library Descriptor File)
- 3、在 web.xml 文件中配置元素 4.在 JSP 文件中引入标签库

五、**TagSupport** 类简介

- 1、处理标签的类必须扩展 javax.servlet.jsp.TagSupport。
- 2、TagSupport 类的主要属性：
 - A.parent 属性：代表嵌套了当前标签的上层标签的处理类；
 - B.pageContext 属性：代表 Web 应用中的 javax.servlet.jsp.PageContext 对象；
- 3、JSP 容器在调用 doStartTag 或者 doEndTag 方法前，会先调用 setPageContext 和 setParent 方法，设置 pageContext 和 parent。因此在标签处理类中可以直接访问 pageContext 变量；
- 4、在 TagSupport 的构造方法中不能访问 pageContext 成员变量，因为此时 JSP 容器还没有调用 setPageContext 方法对 pageContext 进行初始化。

六、**TagSupport** 处理标签的方法

- 1、TagSupport 类提供了两个处理标签的方法：

```
public int doStartTag() throws JspException  
public int doEndTag() throws JspException
```


2、doStartTag: 但 JSP 容器遇到自定义标签的起始标志, 就会调用 doStartTag() 方法。

doStartTag()方法返回一个整数值, 用来决定程序的后续流程。

A.Tag.SKIP_BODY: 表示?>...之间的内容被忽略;

B.Tag.EVAL_BODY_INCLUDE: 表示标签之间的内容被正常执行。

3、doEndTag: 但 JSP 容器遇到自定义标签的结束标志, 就会调用 doEndTag() 方法。doEndTag()方法也返回一个整数值, 用来决定程序后续流程。

A.Tag.SKIP_PAGE: 表示立刻停止执行网页, 网页上未处理的静态内容和 JSP 程序均被忽略任何已有的输出内容立刻返回到客户的浏览器上。

B.Tag_EVAL_PAGE: 表示按照正常的流程继续执行 JSP 网页。

下面来看一个实例:

1、修改 web.xml 文件, 增加自定义标签支持。

[html] [view plaincopyprint?](#)

```
1. <jsp-config>
2.
3. <taglib>
4.
5. <taglib-uri>/tld/helloworld</taglib-uri>
6.
7. <taglib-location>/WEB-INF/tlds/helloworld.tld</taglib-location>
8.
9. </taglib>
10.
11.</jsp-config>
```

2、创建标签库 TLD 文件 tlds\helloworld.tld 。

[html] [view plaincopyprint?](#)

1. `<?xml version= encoding= ?>`
- 2.
3. `<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"`
- 4.
5. `"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">`
- 6.
7. `<taglib>`
- 8.
9. `<tlib-version>1.0</tlib-version><!-- 标签库的版本 -->`
- 10.
11. `<jsp-version>1.2</jsp-version><!-- 这个标签库要求的 JSP 规范版本 -->`
- 12.
13. `<short-name>mytag</short-name><!-- JSP 页面编写工具可以用来创建助记名的可选名字 -->`
- 14.
15. `<tag>`
- 16.
17. `<name>helloworld</name><!-- 唯一标签名 -->`
- 18.
19. `<tag-class>com.yd.mytag.HelloWorldTag</tag-class><!-- 标签 HelloWorldTag 类的完全限定名 -->`
- 20.
21. `<body-content>empty</body-content><!-- 正文内容类型 -->`
- 22.
23. `</tag>`
- 24.
25. `</taglib>`

这里注意：web.xml 和 xxx.tld 这两个 XML 文件的头信息的版本匹配很重要，否则会导致页面报错找不到标签。

3、创建标签处理程序类 HelloWorldTag（重写 doStartTag 和 doEndTag 方法）。

[html] [view plaincopyprint?](#)

```
1. package com.yd.mytag;
2.
3.
4. import java.io.IOException;
5.
6. import javax.servlet.jsp.JspException;
7.
8. import javax.servlet.jsp.JspTagException;
9.
10. import javax.servlet.jsp.tagext.TagSupport;
11.
12. /**
13.
14.  * TagSupport 与 BodyTagSupport 的区别:
15.
16.  * 主要看标签处理类是否要读取标签体的内容和改变标签体返回的内容, 如果不需要
    就用 TagSupport, 否则就用 BodyTagSupport
17.
18.  * 用 TagSupport 实现的标签, 都可以用 BodyTagSupport 来实现, 因为
    BodyTagSupport 继承了 TagSupport
19. */
20.
21. public class HelloWorldTag extends TagSupport {
22.
23.     private static final long serialVersionUID =
24.
25.     @Override
26.
27.     public int doStartTag() throws JspException { //这个方法不用所以直接返回值
28.
29.         return EVAL_BODY_INCLUDE;
30.
31.     }
32.
```

```

33.  @Override
34.
35.  public int doEndTag() throws JspException { //重点在这个方法上
36.
37.      try {
38.
39.          pageContext.getOut().write("Hello World!"); //标签的返回值
40.
41.      } catch (IOException ex) {
42.
43.          throw new JspTagException("错误");
44.
45.      }
46.
47.      return EVAL_PAGE;
48.
49.  }
50.
51.}

```

补充:

doStartTag()方法是遇到标签开始时会呼叫的方法，其合法的返回值是 **EVAL_BODY_INCLUDE** 与 **SKIP_BODY**,前者表示将显示标签间的文字，后者表示不显示标签间的文字。

doEndTag()方法是在遇到标签结束时呼叫的方法，其合法的返回值是 **EVAL_PAGE** 与 **SKIP_PAGE**，前者表示处理完标签后继续执行以下的 JSP 网页，后者是表示不处理接下来的 JSP 网页。

doAfterBody(),这个方法是在显示完标签间文字之后呼叫的，其返回值有 **EVAL_BODY_AGAIN** 与 **SKIP_BODY**，前者会再显示一次标签间的文字，后者则继续执行标签处理的下一步。

EVAL_BODY_INCLUDE: 把 Body 读入存在的输出流中, doStartTag()函数可用。

EVAL_PAGE: 继续处理页面, doEndTag()函数可用。

SKIP_BODY: 忽略对 Body 的处理, doStartTag()和 doAfterBody()函数可用。

SKIP_PAGE: 忽略对余下页面的处理, doEndTag()函数可用。

EVAL_BODY_BUFFERED: 申请缓冲区, 由 setBodyContent()函数得到的 BodyContent 对象来处理 tag 的 body, 如果类实现了 BodyTag, 那么 doStartTag()可用, 否则非法。

(二十一) java 过滤器和监听器详解

过滤器

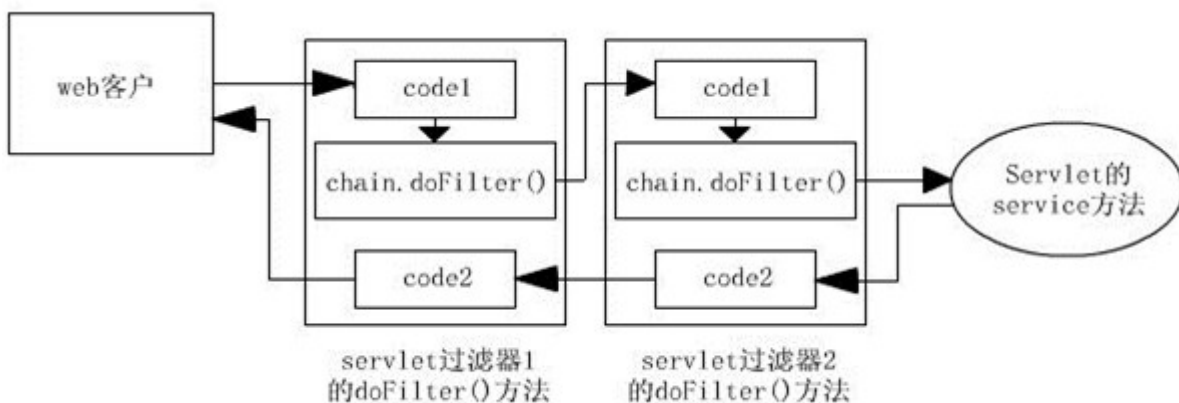
1、Filter 工作原理（执行流程）

当客户端发出 Web 资源的请求时, Web 服务器根据应用程序配置文件设置的过滤规则进行检查, 若客户请求满足过滤规则, 则对客户请求 / 响应进行拦截, 对请求头和请求数据进行检查或改动, 并依次通过过滤器链, 最

后把请求 / 响应交给请求的 **Web** 资源处理。请求信息在过滤器链中可以被修改，也可以根据条件让请求不发往资源处理器，并直接向客户机发回一个响应。当资源处理器完成了对资源的处理后，响应信息将逐级逆向返回。同样在这个过程中，用户可以修改响应信息，从而完成一定的任务。

上面说了，当一个请求符合某个过滤器的过滤条件时该请求就会交给这个过滤器去处理。那么当两个过滤器同时过滤一个请求时谁先谁后呢？这就涉及到了过滤链 **FilterChain**。

所有的奥秘都在 **Filter** 的 **FilterChain** 中。服务器会按照 **web.xml** 中过滤器定义的先后循序组装成一条链，然后一次执行其中的 **doFilter()** 方法。执行的顺序就如下图所示，执行第一个过滤器的 **chain.doFilter()** 之前的代码，第二个过滤器的 **chain.doFilter()** 之前的代码，请求的资源，第二个过滤器的 **chain.doFilter()** 之后的代码，第一个过滤器的 **chain.doFilter()** 之后的代码，最后返回响应。



这里还有一点想补充：大家有没有想过，上面说的“执行请求的资源”究竟是怎么执行的？对于“执行第一个过滤器的 **chain.doFilter()** 之前的代码，第

二个过滤器的 `chain.doFilter()` 之前的代码”这些我可以理解，无非就是按顺序执行一句句的代码，但对于这个“执行请求的资源”我刚开始却是怎么也想不明白。其实是这样的：

通常我们所访问的资源是一个 `servlet` 或 `jsp` 页面，而 `jsp` 其实是一个被封装了的 `servlet`，于是我们就可以统一地认为我们每次访问的都是一个 `Servlet`，而每当我们访问一个 `servlet` 时，`web` 容器都会调用该 `Servlet` 的 `service` 方法去处理请求。而在 `service` 方法又会根据请求方式的不同

（`Get/Post`）去调用相应的 `doGet()` 或 `doPost()` 方法，实际处理请求的就是这个 `doGet` 或 `doPost` 方法。写过 `servlet` 的朋友都应该知道，我们在 `doGet`（或 `doPost`）方法中是通过 `response.getWriter()` 得到客户端的输出流对象，然后用此对象对客户进行响应。

到这里我们就应该理解了过滤器的执行流程了：执行第一个过滤器的 `chain.doFilter()` 之前的代码——>第二个过滤器的 `chain.doFilter()` 之前的代码——>.....——>第 `n` 个过滤器的 `chain.doFilter()` 之前的代码——>所请求 `servlet` 的 `service()` 方法中的代码——>所请求 `servlet` 的 `doGet()` 或 `doPost()` 方法中的代码——>第 `n` 个过滤器的 `chain.doFilter()` 之后的代码——>.....——>第二个过滤器的 `chain.doFilter()` 之后的代码——>第一个过滤器的 `chain.doFilter()` 之后的代码。

过滤器生命周期的四个阶段：

- 1、实例化：Web 容器在部署 Web 应用程序时对所有过滤器进行实例化。Web 容器回调它的无参构造方法。
- 2、初始化：实例化完成之后，马上进行初始化工作。Web 容器回调 `init()` 方法。

3、过滤：请求路径匹配过滤器的 URL 映射时。Web 容器回调 `doFilter()` 方法——主要的工作方法。

4、销毁：Web 容器在卸载 Web 应用程序前，Web 容器回调 `destroy()` 方法。

Servlet 过滤器开发步骤：

1、创建实现 `javax.servlet.Filter` 接口的类。

2、过滤器的 xml 配置。

Servlet 过滤器 API

Servlet 过滤器 API 包含了 3 个接口，它们都在 `javax.servlet` 包中，分别是 `Filter` 接口、`FilterChain` 接口和 `FilterConfig` 接口。

`public Interface Filter`

所有的过滤器都必须实现 `Filter` 接口。该接口定义了 `init`, `doFilter`, `destroy()`

三个方法：

(1) `public void init (FilterConfig filterConfig)`

当开始使用 servlet 过滤器服务时，Web 容器调用此方法一次，为服务准备过滤器；然后在需要使用过滤器的时候调用 `doFilter()`，传送给此方法的 `FilterConfig` 对象，包含 servlet 过滤器的初始化参数。

(2) `public void doFilter(ServletRequest request,
ServletResponse response, FilterChain chain)`

每个过滤器都接受当前的请求和响应，且 `FilterChain` 过滤器链中的过滤器（应该都是符合条件的）都会被执行。`doFilter` 方法中，过滤器可以对请求和响应做它想做的一切，通过调用他们的方法收集数据，或者给对象添加新的行为。过滤器通过传送至 此方法的 `FilterChain` 参数，调用 `chain.`

doFilter() 将控制权传送给下一个过滤器。当这个调用返回后，过滤器可以在它的 **Filter** 方法的最后对响应做些其他的工作。如果过滤器想要终止请求的处理或得到对响应的完全控制，则可以不调用下一个过滤器，而将其重定向至其它一些页面。当链中的最后一个过滤器调用 **chain.doFilter()** 方法时，将运行最初请求的 **Servlet**。

(3) **public void destroy()**

一旦 **doFilter()** 方法里的所有线程退出或已超时，容器调用此方法。服务器调用 **destroy()** 以指出过滤器已结束服务，用于释放过滤器占用的资源。

public interface FilterChain

public void doFilter(ServletRequest request, ServletResponse response)

此方法是由 **Servlet** 容器提供给开发者的，用于对资源请求过滤链的依次调用，通过 **FilterChain** 调用过滤链中的下一个过滤器，如果是最后一个过滤器，则下一个就调用目标资源。

public interface FilterConfig

FilterConfig 接口检索过滤器名、初始化参数以及活动的 **Servlet** 上下文。该接口提供了以下 4 个方法：

(1) **public java.lang.String getFilterName()**

返回 **web.xml** 部署文件中定义的该过滤器的名称。

(2) **public ServletContext getServletContext()**

返回调用者所处的 **servlet** 上下文。

(3) **public java.lang.String getInitParameter(java.lang.String name)**

返回过滤器初始化参数值的字符串形式，当参数不存在时，返回 **null**。**name** 是初始化参数名。

(4)public java.util Enumeration getInitParameterNames()

以 Enumeration 形式返回过滤器所有初始化参数值，如果没有初始化参数，返回为空。

三、应用实例

从上面分析可知，实现 Servlet 过滤器，需要两步：第一步开发过滤器，设计一个实现 Filter 接口的类；第二步通过 web.xml 配置过滤器，实现过滤器和 Servlet、JSP 页面之间的映射。以下设计一个简单的 IP 地址过滤器，根据用户的 IP 地址进行对网站的访问控制。

(1)过滤器的设计 ipfilter.java

[java] [view plaincopyprint?](#)

```
1. package ipf;
2. import java. io. IOException;
3. import javax. servlet. *;
4. public class ipfilter implements Filter // 实现 Filter 接口
5. {protected FilterConfig config;
6. protected String rejectedIP;
7. public void init(FilterConfig filterConfig)throws
8. ServletException
9.
10.{this. config=filterConfig; // 从 Web 务器获取过滤器配置对象
11.rejectedIP=config. getInitParameter( "RejectedIP");
12.//从配置中取得过滤 IP
13.}
14.public void doFilter(ServletRequest request,
15.ServletResponse response. FilterChain chain)throws
16.IOException, ServletException
17.{RequestDispatcher dispatcher=request.getRequestDispatcher("");
18.String remoteIP=request. getRemoteAddr(); //获取客户请求 IP
19.int i=remoteIP. lastIndexOf(". ");
20.int r=rejectedIP. lastIndexOf(". ");
```

```

21.String relPscope=rejectedIP. substring(0, r); // 过滤 IP 段
22.if(relPscope. equals(remotelP. substring(0. i)))
23.{    dispatcher. forward(request, response); //重定向到 rejectedError. jsp 页面
24.    return; // 阻塞，直接返 Web 回客户端
25.}
26.else{chain. doFilter(request, response); //调用过滤链上的下一个过滤器
27.}
28.}
29.public void destroy()

```

//过滤器功能完成后，由 Web 服务器调用执行，回收过滤器资源

注意：chain. doFilterO 语句以前的代码用于对客户请求的处理；以后的代码用于对响应进行处理。

(2)配置过滤器

在应用程序 Web—INF 目录下的 web. xml 描述符文件中添加以下代码：

[html] [view plaincopyprint?](#)

```

1. <filter>
2. <filter-name>ipfilter< / filter-name> // 过滤器名称
3. <filter-class>ipf. ipfilter< / filter-class> // 实现过滤器的类
4. <init—param>
5. <param—name>RejectedIP< / param-name> // 过滤器初始化参数名
   RejectedIP
6. <param-value>192.168.12.* / param-value>
7. < / init—pamm>
8. < / filter>
9. <filter-mapping> // 过滤器映射(规律规则)
10.<filter-name>ipfilter< / filter-name>
11.<url—patten>/*< / ud-patten>
12. // 映射到 Web 应用根目录下的所有 JSP 文件

```

通过以上设计与配置，就禁止了 IP 地址处在 192.168.12 网段的用户对网站的访问。

监听器

一、监听器概述

监听你的 web 应用，监听许多信息的初始化，销毁，增加，修改，删除值等

Servlet 监听器用于监听一些重要事件的发生，监听器对象可以在事情发生前、发生后可以做一些必要的处理。

1. **Listener** 是 **Servlet** 的监听器
2. 可以监听客户端的请求、服务端的操作等。
3. 通过监听器，可以自动激发一些操作，如监听在线用户数量,当增加一个 **HttpSession** 时，给在线人数加 1。
4. 编写监听器需要实现相应的接口
5. 编写完成后在 **web.xml** 文件中配置一下,就可以起作用了
6. 可以在不修改现有系统基础上,增加 web 应用程序生命周期事件的跟踪

servlet 规范中为每种事件监听器都定义了相应的接口,在编写事件监听器程序时只需实现这些接口就可以了。一些 **Servlet** 事件监听器需要在 web 应用程序的部署 文件描述符文件（**web.xml**）中进行注册(注册之

后才能发布), 一个 **web.xml** 可以注册多个 **servlet** 事件监听器。**web** 服务器按照它们在 **web.xml** 中注册顺序来加载和注册这些 **servlet** 事件监听器。**servlet** 事件监听器的注册和调用过程都是由 **web** 容器自动完成的, 当发生被监听对象被创建, 修改, 销毁等事件时, **web** 容器将调用与之相关的 **servlet** 事件监听器对象的相应方法(所监听到的对象如果在创建、修改、销毁事件触发的时候就会调用这些监听器这就相当于面向事件编程的概念), 用户在这些方法中编写的事件处理代码(相当于 JS 中的事件响应)即被执行。由于在一个 **web** 应用程序中只会为每个事件监听器类创建一个实例对象, 有可能出现多个线程同时调用一个事件监听对象的情况, 所以要注意多线程安全问题。

二、监听器类型

按监听的对象划分: **servlet2.4** 规范定义的事件有三种:

- 1.用于监听应用程序环境对象 (**ServletContext**) 的事件监听器
- 2.用于监听用户会话对象 (**HttpSession**) 的事件监听器
- 3.用于监听请求消息对象 (**ServletRequest**) 的事件监听器

按监听的事件类项划分

- 1.用于监听域对象自身的创建和销毁的事件监听器
- 2.用于监听域对象中的属性的增加和删除的事件监听器
- 3.用于监听绑定到 **HttpSession** 域中的某个对象的状态的事件监听器

在一个 **web** 应用程序的整个运行周期内, **web** 容器会创建和销毁三个重要的对象, **ServletContext**, **HttpSession**, **ServletRequest**。

PS: 其中 **Context** 为 JSP 页面包装页面的上下文.由容器创建和初始化,管理对属于 JSP 中特殊可见部分中已命名对象的访问. 该接口用来定义了一个 **Servlet** 的环境对象。也可认为这是多个客户端共享的信息，它与 **session** 的区别在于应用范围的不同，**session** 只对应于一个用户。

servlet2.4 中定义了三个接口：

ServletContextListener, HttpSessionListener, ServletRequestListener。

分别实现对应的接口就可以实现对应的监听处理

在 **ServletContextListener** 接口中定义了两个事件处理方法，分别是

contextInitialized () 和 **contextDestroyed ()**

public void contextInitialized(ServletContextEvent sce)

这个方法接受一个 **ServletContextEvent** 类型参数，在 **contextInitialized**

可以通过这个参数获得当前被创建的 **ServletContext** 对象。

public void contextDestroyed(ServletContextEvent sce)

2.在 **HttpSessionListneter** 接口中共定义了两个事件处理方法，分别是

sessionCreated () 和 **sessionDestroyed ()**

public void sessionCreated(HttpSessionEvent se)

这个方法接受一个(**HttpSessionEvent** 类型参数，在 **sessionCreated** 可

以通过这个参数获得当前被创建的 **HttpSession** 对象。

public void sessionDestroyed(HttpSessionEvent se)

3.在 **ServletRequestListener** 接口中定义了两个事件处理方法，分别是

requestInitialized () 和 **requestDestroyed ()**

public void requestInitialized(ServletRequestEvent sre)

这个方法接受一个(**ServletRequestEvent** 类型参数，在

`requestInitialized` 可以通过这个参数获得当前被创建的 `ServletRequest` 对象。

```
public void requestDestroyed(ServletRequestEvent sre)
```

可以看出三个监听器接口中定义的方法非常相似，执行原理与应用方式也相似，在 `web` 应用程序中可以注册一个或者多个实现了某一接口的事件监听器，`web` 容器在创建或销毁某一对象（如 `ServletContext`, `HttpSession`）时就会产生相应的事件对象

（如 `ServletContextEvent`，或者 `HttpSessionEvent`），接着依次调用每个事件监听器中的相应处理方法，并将产生的事件对象传递给这些方法。

三、分类及介绍

1. `ServletContextListener`: 用于监听 `WEB` 应用启动和销毁的事件，监听器类需要实现 `javax.servlet.ServletContextListener` 接口。
2. `ServletContextAttributeListener`: 用于监听 `WEB` 应用属性改变的事件，包括：增加属性、删除属性、修改属性，监听器类需要实现 `javax.servlet.ServletContextAttributeListener` 接口。
3. `HttpSessionListener`: 用于监听 `Session` 对象的创建和销毁，监听器类需要实现 `javax.servlet.http.HttpSessionListener` 接口或者 `javax.servlet.http.HttpSessionActivationListener` 接口，或者两个都实现。
4. `HttpSessionActivationListener`: 用于监听 `Session` 对象的钝化/活化事件，监听器类需要实现 `javax.servlet.http.HttpSessionListener` 接口

或者 `javax.servlet.http.HttpSessionActivationListener` 接口，或者两个都实现。

5. `HttpSessionAttributeListener`: 用于监听 `Session` 对象属性的改变事件，监听器类需要实现

`javax.servlet.http.HttpSessionAttributeListener` 接口。

四、部署

监听器的部署在 `web.xml` 文件中配置，在配置文件中，它的位置应该在过滤器的后面 `Servlet` 的前面

五、示例

第一步：编写监听器类

[html] [view plaincopyprint?](#)

```
1. package cn.listen;
2.
3. import javax.servlet.ServletContextEvent;
4.
5. import javax.servlet.ServletContextListener;
6.
7.
8. public class MyListener implements ServletContextListener {
9.
10.     public void contextDestroyed(ServletContextEvent sce) {
11.
12.         System.out.println("die");
13.
14.     }
15.
16.     public void contextInitialized(ServletContextEvent sce) {
17.
18.         System.out.println("init");
19.
20.     }
21.
```


22. }

第二步：布置安装

```
<listener>  
  <listener-class>cn.listen.MyListener</listener-class>  
</listener>
```

运行服务器会出现

```
[20:42:38.406] {main} WebApp[http://default] active  
init  
[20:42:38.437] {main} WebApp[http://default/MyProj] active  
监听到了应用启动。
```

（二十二）华山论 session 和 cookie 机制

会话（Session）跟踪是 Web 程序中常用的技术，用来跟踪用户的整个会话。常用的会话跟踪技术是 Cookie 与 Session。Cookie 通过在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份。

一. cookie 和 session 机制之间的区别和联系

具体来说 cookie 机制采用的是在客户端保持状态的方案。它是在用户端的会话状态的存贮机制，他需要用户打开客户端的 cookie 支持。cookie 的作用就是为了解决 HTTP 协议无状态的缺陷所作的努力.而 session 机制采用的是一种在客户端与服务器之间保持状态的解决方案。同时我们也看到，由于采用服务器端保持状态的方案在客户端也需要保存一个标识，所以 session 机制可能需要借助于 cookie 机制来达到保存标识的目的。而 session 提供了方便管理全局变量的方式

session 是针对每一个用户的，变量的值保存在服务器上，用一个 sessionID 来区分是哪个用户 session 变量,这个值是通过用户的浏览器在访问的时候返回给服务器，当客户禁用 cookie 时，这个值也可能设置为由 get 来返回给服务器。就安全性来说：当你访问一个使用 session 的站点，同时在自己机子上建立一个 cookie，建议在服务器端的 SESSION 机制更安全些.因为它不会任意读取客户存储的信息。

正统的 cookie 分发是通过扩展 HTTP 协议来实现的，服务器通过在 HTTP 的响应头中加上一行特殊的指示以提示浏览器按照指示生成相应的

cookie。从网络服务器观点看所有 HTTP 请求都独立于先前请求。就是说每一个 HTTP 响应完全依赖于相应请求中包含的信息。状态管理机制克服了 HTTP 的一些限制并允许网络客户端及服务器端维护请求间的关系。在这种关系维持的期间叫做会话(session)。

Cookies 是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器。IETF RFC 2965 HTTP State Management Mechanism 是通用 cookie 规范。网络服务器用 HTTP 头向客户端发送 cookies，在客户端，浏览器解析这些 cookies 并将它们保存为一个本地文件，它会自动将同一服务器的任何请求缚上这些 cookies

二.理解 session 机制

session 机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。当程序需要为某个客户端的请求创建一个 session 的时候，服务器首先检查这个客户端的请求里是否已包含了一个 session 标识——称为 session id，如果已包含一个 session id 则说明以前已经为此客户端创建过 session，服务器就按照 session id 把这个 session 检索出来使用（如果检索不到，可能会新建一个），如果客户端请求不包含 session id，则为此客户端创建一个 session 并且生成一个与此 session 相关联的 session id，session id 的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个 session id 将被在本次响应中返回给客户端保存。

保存这个 session id 的方式可以采用 cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器。一般这个 cookie 的名字都是类似

于 **SESSIONID**，而。比如 **weblogic** 对于 **web** 应用程序生成的 **cookie**，
JSESSIONID= ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764，它的名字就是 **JSESSIONID**。由于 **cookie** 可以被人为的禁止，必须有其他机制以便在 **cookie** 被禁止时仍然能够把 **session id** 传递回服务器。经常被使用的一种技术叫做 **URL** 重写，就是把 **session id** 直接附加在 **URL** 路径的后面，附加方式也有两种，一种是作为 **URL** 路径的附加信息，表现形式为

http://...../xxx;jsessionid= ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764

另一种是作为查询字符串附加在 **URL** 后面，表现形式为

http://...../xxx?jsessionid=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764

这两种方式对于用户来说是没有区别的，只是服务器在解析的时候处理的方式不同，采用第一种方式也有利于把 **session id** 的信息和正常程序参数区分开来。为了在整个交互过程中始终保持状态，就必须在每个客户端可能请求的路径后面都包含这个 **session id**。

另一种技术叫做表单隐藏字段。就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把 **session id** 传递回服务器。这种技术现在已较少应用，笔者接触过的很古老的 **iPlanet6(SunONE 应用服务器的前身)** 就使用了这种技术。实际上这种技术可以简单的用对 **action** 应用 **URL** 重写来代替。

在谈论 **session** 机制的时候，常常听到这样一种误解“只要关闭浏览器，**session** 就消失了”。其实可以想象一下会员卡的例子，除非顾客主动对店家

提出销卡，否则店家绝对不会轻易删除顾客的资料。对 **session** 来说也是一样的，除非程序通知服务器删除一个 **session**，否则服务器会一直保留，程序一般都是在用户做 **log off** 的时候发个指令去删除 **session**。然而浏览器从来不会主动在关闭之前通知服务器它将要关闭，因此服务器根本不会有机会知道浏览器已经关闭，之所以会有这种错觉，是大部分 **session** 机制都使用会话 **cookie** 来保存 **session id**，而关闭浏览器后这个 **session id** 就消失了，再次连接服务器时也就无法找到原来的 **session**。如果服务器设置的 **cookie** 被保存到硬盘上，或者使用某种手段改写浏览器发出的 **HTTP** 请求头，把原来的 **session id** 发送给服务器，则再次打开浏览器仍然能够找到原来的 **session**。

恰恰是由于关闭浏览器不会导致 **session** 被删除，迫使服务器为 **session** 设置了一个失效时间，当距离客户端上一次使用 **session** 的时间超过这个失效时间时，服务器就可以认为客户端已经停止了活动，才会把 **session** 删除以节省存储空间。

由 **JSESSIONID** 谈 **cookie** 与 **SESSION** 的区别和联系

在一些投票之类的场合，我们往往因为公平的原则要求每人只能投一票，在一些 **WEB** 开发中也有类似的情况，这时候我们通常会使用 **COOKIE** 来实现，例如如下的代码：

[html] [view plaincopyprint?](#)

1. `<% cookie[cookies = .getCookies();`
2. `if (cookies.lenght == 0 || cookies == null)`
3. `doStuffForNewbie();`
4. `//没有访问过`

```

5. }else
6. {
7. doStuffForReturnVisitor(); //已经访问过了
8. }% >

```

这是很浅显易懂的道理，检测 **COOKIE** 的存在，如果存在说明已经运行过写入 **COOKIE** 的代码了，然而运行以上的代码后，无论何时结果都是执行 **doStuffForReturnVisitor()**，通过控制面板-Internet 选项-设置-察看文件却始终看不到生成的 **cookie** 文件，奇怪，代码明明没有问题，不过既然有 **cookie**，那就显示出来看看。

[html] [view plaincopyprint?](#)

```

1. cookie[]cookies = .getCookies();
2. if (cookies.lenght == 0 || cookies == null)
3. out.println("Has not visited this website");
4. }else{
5. for (int i =  ; i < cookie.length; i++){
6. out.println("cookie name:" + cookies[i].getName() + "cookie value:" +
7. cookie[i].getValue());}}

```

运行结果:

cookie name:JSESSIONID cookie value:KWJHUG6JJM65HS2K6 为什么会有 **cookie** 呢,大家都知道，**http** 是无状态的协议，客户每次读取 **web** 页面时，服务器都打开新的会话，而且服务器也不会自动维护客户的上下文信息，那么要怎么才能实现网上商店中的购物车呢，**session** 就是一种保存上下文信息的机制，它是针对每一个用户的，变量的值保存在服务器端，通过

SessionID 来区分不同的客户,session 是以 cookie 或 URL 重写为基础的,默认使用 cookie 来实现,系统会创建一个名为 JSESSIONID 的输出 cookie,我们叫做 session cookie,以区别 persistent cookies,也就是我们通常所说的 cookie,注意 session cookie 是存储于浏览器内存中的,并不是写到硬盘上的,这也就是我们刚才看到的 JSESSIONID,我们通常情是看不到 JSESSIONID 的,但是当我们把浏览器的 cookie 禁止后,web 服务器会采用 URL 重写的方式传递 Sessionid,我们就可以在地址栏看到 sessionid=KWJHUG6JJM65HS2K6 之类的字符串。

明白了原理,我们就可以很容易的分辨出 persistent cookies 和 session cookie 的区别了,网上那些关于两者安全性的讨论也就一目了然了,session cookie 针对某一次会话而言,会话结束 session cookie 也就随着消失了,而 persistent cookie 只是存在于客户端硬盘上的一段文本(通常是加密的),而且可能会遭到 cookie 欺骗以及针对 cookie 的跨站脚本攻击,自然不如 session cookie 安全了。

通常 session cookie 是不能跨窗口使用的,当你新开了一个浏览器窗口进入相同页面时,系统会赋予你一个新的 sessionid,这样我们信息共享的目的就达不到了,此时我们可以先把 sessionid 保存在 persistent cookie 中,然后在新窗口中读出来,就可以得到上一个窗口 SessionID 了,这样通过 session cookie 和 persistent cookie 的结合我们就实现了跨窗口的 session tracking(会话跟踪)。

在一些 web 开发的书中,往往只是简单的把 Session 和 cookie 作为两种并列的 http 传送信息的方式,session cookies 位于服务器端,

persistent cookie 位于客户端，可是 **session** 又是以 **cookie** 为基础的，明白的两者之间的联系和区别，我们就不难选择合适的技术来开发 **web service** 了。

（二十三）常见乱码解决以及 javaBean 基础知识

乱码问题应该是做 javaWeb 开发人员都遇到过的问题吧，这个问题当时还影响了我学习 java 的想法，甚至有过想放弃的想法，没办法，当时年轻，呵呵。其实产生乱码问题的原因有很多，解决乱码的问题也有很多，现在就一一来看一下：

出现乱码的地方大致可以分为以下三种：

- 1 jsp 页面中
- 2 jsp 页面之间相互传参的参数
- 3 与数据库中数据的存取

解决方案大致可以分为三种：

- 1 出现在 jsp 页面中，是由于没有设置 jsp 页面的中文字符编码。
- 2 出现在 jsp 页面之间相互传参，是由于参数没有设置正确的字符编码。
- 3 以上 2 个问题解决了，那么存到数据库中，自然就不存在乱码。除非你对存入到数据库里的数据再次进行编码。

具体的解决方法：

1.在表单页面头部设置字符编码为 utf-8

```
<%@ page language="java" import="java.util.*" pageEncoding="utf-8"%>
```

加上这句解决 jsp 页面中的中文乱码显示,tomcat 编译完后向客户端输出的 html 文件不是采用中文编码，所以会导致乱码产生。

2.设置页面请求和回应的编码：


```
<%request.setCharacterEncoding("utf-8");  
response.setCharacterEncoding("utf-8"); %>
```

加上这句解决 jsp 页面中的中文参数传递乱码。把浏览器默认使用的编码设置为“UTF-8”发送请求参数。

3.String(request.getParameter("name").getBytes("ISO8859_1"),"utf-8");这句的意思是，把传来的参数全部编码转换成 utf-8，这样做的缺点是每次传来一个参数都要这样写，很麻烦。

同样可通过设置 server.xml 配置文件来实现。

[html] [view plaincopyprint?](#)

1. **< Connector**
2. **port =**
3. **maxHttpHeaderSize =**
4. **maxThreads =**
5. **minSpareThreads =**
6. **maxSpareThreads =**
7. **enableLookups =**
8. **redirectPort =**
9. **acceptCount =**
10. **connectionTimeout =**
11. **disableUploadTimeout =**
12. **URIEncoding =** 

但是这样就应用到整个 webapp 中去了。

4.还可以修改 web.xml 文件，配置一个过滤器。其原理都一样，只是换种方式而已

1.编写过滤器类：

[html] [view plaincopyprint?](#)

1. package org.RN.util;
- 2.

```

3. import java.io.IOException;
4.
5. import javax.servlet.Filter;
6. import javax.servlet.FilterChain;
7. import javax.servlet.FilterConfig;
8. import javax.servlet.ServletException;
9. import javax.servlet.ServletRequest;
10. import javax.servlet.ServletResponse;
11.
12. public class Encoding implements Filter {
13.     @SuppressWarnings("unused")
14.     private FilterConfig config=    ;
15.     String encoding=    ;
16.     public void destroy() {
17.         this.encoding=    ;
18.         this.config=    ;
19.
20.     }
21.
22.     public void doFilter(ServletRequest request, ServletResponse response,
23.         FilterChain chain) throws IOException, ServletException {
24.         if(encoding!=null)
25.             request.setCharacterEncoding(encoding);
26.         chain.doFilter(request, response);
27.
28.
29.     }
30.
31.     public void init(FilterConfig arg0) throws ServletException {
32.         this.config=    ;
33.         this.encoding=    .getInitParameter("encoding");
34.
35.     }
36.
37. }

```

2.在 web.xml 配置

[html] [view plaincopyprint?](#)

1. `<filter>`
2. `<description>缓存过滤</description>`
3. `<filter-name>Encoding</filter-name>`
4. `<filter-class>`
5. `filter.Encoding`
6. `</filter-class>`
7. `</filter>`
8. `<filter-mapping>`
9. `<filter-name>Encoding</filter-name>`
10. `<url-pattern>*</url-pattern>`
11. `</filter-mapping>`

[html] [view plaincopyprint?](#)

1. `</filter>`

5.还有一种常见的乱码问题就是下载时出现文件名乱码

原来处理下载的代码如下：

```
response.setHeader("Content-Disposition", "attachment; filename=" + java.  
net.URLEncoder.encode(fileName, "UTF-8"));
```

URLEncoder 类包含将字符串转换为

application/x-www-form-urlencoded MIME 格式的静态方法。

URLDecoder 与 URLEncoder 类相对应的 URLDecoder 类有两种静态方法。它们解码以 *x-www-form-urlencoded* 这种形式编码的 *string*。

也就是说，它们把所有的加号(+)转换成空格符，把所有的%~~xx~~分别转换成与之相对应的字符：

JavaBean 是一种 **JAVA** 语言写成的可重用组件。为写成 **JavaBean**，类必须是具体的和公共的，并且具有无参数的构造器。**JavaBean** 通过提供符合一致性设计模式的公共方法将内部域暴露成员属性。众所周知，属性名称符合这种模式，其他 **Java** 类可以通过自身机制发现和操作这些 **JavaBean** 属性。

JavaBean 的任务就是：“Write once, run anywhere, reuse everywhere”，即“一次性编写，任何地方执行，任何地方重用”。这个任何实际上就是要解决困扰软件工业的日益增加的复杂性，提供一个简单的、紧凑的和优秀的问题解决方案。

JavaBean 的范围 **Scope** 是一个具有生命时间的变量。**JavaBean** 的范围在<jsp:useBean scope="....">标志中右边进行表示。将产生一个 **JavaBean** 的快捷参考。说明：jsp 服务器引擎将剥离<jsp。。。。标记。并且在最终用户的浏览器上无法显示实际代码。

存在下面四种范围：页面 page、请求 request、对话 session、应用 application。

对话范围：

对话范围的 **JavaBean** 主要应用于跨多个页面和时间段：例如填充 用户信息。 添加信息并且接受回馈，保存用户最近执行页面的轨迹。对话范围 **JavaBean** 保留一些和用户对话 ID 相关的信息。这些信息来自临时的对

话 cookie，并在当用户关闭浏览器时，这个 cookie 将从客户端和服务端删除。

页面/请求范围：

页面和请求范围的 JavaBean 有时类似表单的 bean，这是因为他们大都用于处理表单。表单需要很长的时间来处理用户的输入，通常情况下用于页面接受 HTTP/POST 或者 GET 请求。另外页面和请求范围的 bean 可以用于减少大型站点服务器上的负载，如果使用对话 bean，耽搁的处理可能会消耗掉很多资源。

应用：

应用范围通常应用于服务器的部件，例如 JDBC [连接池](#)、应用监视、用户计数和其他参与用户行为的类。在 Bean 中限制 HTML 的产生：理论上，JavaBean 将不会产生任何 HTML，因为这是 jsp 层负责的工作；然而，为动态消息提供一些预先准备的格式是非常有用的。产生的 HTML 将被标注的 JavaBean 方法返回。

（二十四）Xml 基础详解和 DTD 验证

Xml 基础详解

Xml:可扩展标记语言 (Extensible Markup Language, XML)，用于标记电子文件使其具有结构性的标记语言，可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。XML 是标准通用标记语言 (SGML) 的子集，非常适合 Web 传输。XML 提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。

Xml 的基本语法：

- 1 任何的起始标签都必须有一个结束标签。
- 2 可以采用另一种简化语法，可以在一个标签中同时表示起始和结束标签。这种语法是在大于符号之前紧跟一个斜线 (/)，例如<tag/ >。XML 解析器会将其翻译成<tag></tag>。
- 3 标签必须按合适的顺序进行嵌套，所以结束标签必须按镜像顺序匹配起始标签，例如 `this is a samplestring`。这好比是将起始和结束标签看作是数学中的左右括号：在没有关闭所有的内部括号之前，是不能关闭外面的括号的。
- 4 所有的特性都必须有值。
- 5 所有的特性都必须在值的周围加上双引号。
- 6.对于 XML 文档来说，<?处理指令必须要顶格写，前面不能有任何的空白。
7. XML 元素可以具有属性，属性的形式为：
属性名="属性值"，比如 `gender="male"`
属性值需要使用单引号或双引号括起来。多个属性之间使用空格分开。
8. 通过样式，我们可以实现 XML 内容与展现形式的分离

9. XML 的注释：<!-- comments -->，注释不允许嵌套

10. 在一个元素上，相同的属性只能出现一次。

注意以下几点：

1.XML 中的每个元素都是成对出现的（有开始，有结束），

<student> </student>，XML 中的元素嵌套关系要保持正确性，即先开始的标记要先结束，后开始的标记要后结束。

2. 每一个 XML 文档都有且只有一个根元素（Root Element）。所谓根元素，就是唯一一个包含了其他所有元素的元素。

3. XML 描述的是文档的内容与语义，而不是文档应当如何显示。

4.格式正规（well formed）的 XML 文档。符合 XML 语法要求的 XML 文档就是格式正规的 XML 文档。

5. 有效的（valid）XML 文档。首先 XML 文档是个格式正规的 XML 文档，然后又需要满足 DTD 的要求，这样的 XML 文档称为有效的 XML 文档

6. #PCDATA （Parsed Character Data），可解析的字符数据。

XML 文件的验证机制——DTD

文档类型定义（DTD）可定义合法的 XML 文档构建模块。它使用一系列合法的元素来定义文档的结构。

DTD 可被成行地声明于 XML 文档中，也可作为一个外部引用。

内部的 DOCTYPE 声明

假如 DTD 被包含在您的 XML 源文件中，它应当通过下面的语法包装在个 DOCTYPE 声明中：

<!DOCTYPE 根元素 [元素声明]>

带有 DTD 的 XML 文档实例

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)> ]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

以上 DTD 解释如下: !DOCTYPE note (第二行)定义此文档是 note 类型的文档。

!ELEMENT note (第三行)定义 note 元素有四个元素: "to、from、heading、body"

!ELEMENT to (第四行)定义 to 元素为 "#PCDATA" 类型

!ELEMENT from (第五行)定义 from 元素为 "#PCDATA" 类型

!ELEMENT heading (第六行)定义 heading 元素为 "#PCDATA" 类型

!ELEMENT body (第七行)定义 body 元素为 "#PCDATA" 类型

外部文档声明

假如 DTD 位于 XML 源文件的外部, 那么它应通过下面的语法被封装在一个 DOCTYPE 定义中: <!DOCTYPE 根元素 SYSTEM "文件名

"> 这个 XML 文档和上面的 XML 文档相同，但是拥有一个外部的 DTD:

[html] [view plaincopyprint?](#)

1. `<?xml version="1.0" encoding="UTF-8" ?>`
- 2.
3. `<!DOCTYPE note SYSTEM "note.dtd">`
- 4.
5. `<note>`
- 6.
7. `<to>Tove</to>`
- 8.
9. `<from>Jani</from>`
- 10.
11. `<heading>Reminder</heading>`
- 12.
13. `<body>Don't forget me this weekend!</body>`
- 14.
15. `</note>`

这是包含 DTD 的 "note.dtd" 文件:

[html] [view plaincopyprint?](#)

1. `<!ELEMENT note (to,from,heading,body)>`
- 2.
3. `<!ELEMENT to (#PCDATA)>`
- 4.
5. `<!ELEMENT from (#PCDATA)>`
- 6.
7. `<!ELEMENT heading (#PCDATA)>`
- 8.
9. `<!ELEMENT body (#PCDATA)>`

下面来看一下 **DTD** 在 **xml** 每一个模块中的定义:

在一个 DTD 中, 元素通过元素声明来进行声明。

声明一个元素

在 DTD 中, XML 元素通过元素声明来进行声明。元素声明使用下面的语法:

`<!ELEMENT 元素名称 类别>`

或者 `<!ELEMENT 元素名称 (元素内容)>`

空元素

空元素通过类别关键词 **EMPTY** 进行声明:

`<!ELEMENT 元素名称 EMPTY>`

只有 **PCDATA** 的元素

只有 **PCDATA** 的元素通过圆括号中的 **#PCDATA** 进行声明:

`<!ELEMENT 元素名称 (#PCDATA)>`

带有任何内容的元素

通过类别关键词 **ANY** 声明的元素, 可包含任何可解析数据的组合:

`<!ELEMENT 元素名称 ANY>`

带有子元素 (序列) 的元素

带有一个或多个子元素的元素通过圆括号中的子元素名进行声明:

`<!ELEMENT 元素名称 (子元素名称 1)>` 或

者 `<!ELEMENT 元素名称 (子元素名称 1,子元素名称 2,.....)>`

相同的元素至少出现一次的声明

语法格式为: <!ELEMENT element-name (child-name+)>

例: <!ELEMENT note (message+)>

例中的+是指子元素 `message` 必须在被包含的 `note` 元素里出现一次或者多次。

相同的元素不出现或者多次出现的声明

语法格式为:

<!ELEMENT element-name (child-name*)>

例: <!ELEMENT note (message*)>

例中的*是指子元素 `message` 能够在被包含的 `note` 元素里不出现或者出现多次。

属性

在 DTD 中, 属性通过 `ATTLIST` 声明来进行声明。

声明属性

属性声明使用下列语法:

<!ATTLIST 元素名称 属性名称 属性类型 默认值>

以下是属性类型的选项:

类型 描述

`CDATA` 值为字符数据 (character data)

`(en1|en2|..)` 此值是枚举列表中的一个值

`ID` 值为唯一的 `id`

`IDREF` 值为另外一个元素的 `id`

`IDREFS` 值为其他 `id` 的列表

NMTOKEN 值为合法的 XML 名称

NMTOKENS 值为合法的 XML 名称的列表

ENTITY 值是一个实体

ENTITIES 值是一个实体列表

NOTATION 此值是符号的名称

xml: 值是一个预定义的 XML 值

默认值参数可使用下列值:

值 解释

值 属性的默认值

#REQUIRED 属性值是必需的

#IMPLIED 属性不是必需的

#FIXED value 属性值是固定的

规定一个默认的属性值

DTD:

```
<!ELEMENT square EMPTY>
```

```
<!ATTLIST square width CDATA "0">
```

合法的 XML:

```
<square width="100" />
```

在上面的例子中, "square" 被定义为带有 CDATA 类型的 "width" 属性的空元素。如果宽度没有被设定, 其默认值为 0。

实体

实体是用于定义引用普通文本或特殊字符的快捷方式的变量。 实体引用是对实体的引用。

实体可在内部或外部进行声明。

一个内部实体声明

语法:

`<!ENTITY 实体名称 "实体的值">`

DTD 例子:

`<!ENTITY writer "Bill Gates">`

`<!ENTITY copyright "Copyright >`

具体的 url">XML 例子: `<author>&writer;©right;</author>`

注释: 一个实体由三部分构成: 一个和号 (&), 一个实体名称, 以及一个分号 (;)。

一个外部实体声明

语法:

`<!ENTITY 实体名称 SYSTEM "URI/URL">`

DTD 例子:

`<!ENTITY writer SYSTEM "具体的 url">`

`<!ENTITY copyright SYSTEM "具体的 url">XML`

例子: `<author>&writer;©right;</author>`

（二十五）XML 之 Schema 验证

XML Schema 是用一套预先规定的 XML 元素和属性创建的，这些元素和属性定义了 XML 文档的结构和内容模式。XML Schema 规定 XML 文档实例的结构和每个元素/属性的数据类型。

为什么要用 Schema

DTD 的局限性

- 1.DTD 不遵守 XML 语法（写 XML 文档实例时候用一种语法，写 DTD 的时候用另外一种语法）
- 2.DTD 数据类型有限（与数据库数据类型不一致）
- 3.DTD 不可扩展
- 4.DTD 不支持命名空间（命名冲突）

.Schema 的新特性

- 1.Schema 基于 XML 语法
- 2.Schema 可以用能处理 XML 文档的工具处理
- 3.Schema 大大扩充了数据类型，可以自定义数据类型
- 4.Schema 支持元素的继承—Object-Oriented’
- 5.Schema 支持属性组

一：Schema 基础知识

1. Schema（模式）：其作用与 dtd 一样，也是用于验证 XML 文档的有效性，只不过它提供了比 dtd 更强大的功能和更细粒度的数据类型，另外 Schema 还可以自定义数据类型。此外，Schema 也是一个 XML 文件，而 dtd 则不是。

2. 所有的 schema 文档，其根元素必须为 schema。

3.Schema 的文档结构



4.schema 的数据类型

1.基本类型

数据类型	描述
string	表示字符串
boolean	布尔型
decimal	代表特定精度的数字
float	表示单精度32位浮点数
double	表示双精度64位浮点数
duration	表示持续时间
dateTime	代表特定的时间
time	代表特定的时间，但是是每天重复的
date	代表日期
hexBinary	代表十六进制数
anyURI	代表一个URI，用来定位文件
NOTATION	代表 NOTATION类型

2.扩展数据类型

数据类型	描述
ID	用于唯一标识元素
IDREF	参考ID类型的元素或属性
ENTITY	实体类型
NMTOKEN	NMTOKEN类型
NMTOKENS	NMTOKEN类型集
long	表示整型数，大小介于-9223372036854775808和9223372036854775807之间
int	表示整型数，大小介于-2147483648和2147483647之间
short	表示整型数，大小介于-32768和32767之间
byte	表示整型数，大小介于-128和127之间

3.数据类型的特性

特性	描述
enumeration	在指定的数据集中选择，限定用户的选值
fractionDigits	限定最大的小数位，用于控制精度
length	指定数据的长度
maxExclusive	指定数据的最大值（小于）
maxInclusive	指定数据的最大值（小于等于）
maxLength	指定长度的最大值
minExclusive	指定最小值（大于）
minInclusive	指定最小值（大于等于）
minLength	指定最小长度
Pattern	指定数据的显示规范

二：schema 的元素类型

1.schema 元素：

作用：包含已经定义的 schema

用法：<xs:schema>

•属性

–xmlns –targetNamespace

2.element 元素作用：声明一个元素

属性： -name -type -ref -minOccurs -maxOccurs

-substitutionGroup -fixed -default

示例：

[html] view plaincopyprint?

1. `<xs:element name= type= />`
- 2.
3. `<xs:element name= type= />`
- 4.
5. `<xs:element name= >`

3.group 元素

作用：把一组元素声明组合在一起，以便它们能够一起被复合类型应用

•属性： name/ref

示例：

[html] view plaincopyprint?

1. `<xs:group name= >`
- 2.
3. `<xs:sequence>`
- 4.
5. `<xs:element ref= />`
- 6.
7. `<xs:element ref= />`
- 8.
9. `</xs:sequence>`
- 10.
11. `</xs:group>`

4.attribute 元素

作用：声明一个属性

•属性：name/type/ref/use

•示例：

[html] [view plaincopyprint?](#)

```
1. <xs:complexType name=
2.
3. <xs:attribute name= type= use= />
4.
5. </xs:complexType>
```

5.attributeGroup 元素

作用：把一组属性声明组合在一起，以便可以被复合类型应用

.属性：name/ref

.示例：

[html] [view plaincopyprint?](#)

```
1. <xs:attributeGroup name=
2.
3. <xs:attribute name= type= />
4.
5. <xs:attribute name= type= />
6.
7. </xs:attributeGroup>
```

6.simpleType 元素

作用：定义一个简单类型，它决定了元素和属性值的约束和相关信息

.属性：name

.内容：应用已经存在的简单类型，三种方式：

–restrict→限定一个范围

–list→从列表中选择

–union→包含一个值的结合

1.子元素为: <xs:restriction> 定义一个约束条件

2.子元素为: <xs:list>从一个特定数据类型的集合中选择定义一个简单类型元素

3.子元素为: <xs:union>从一个特定简单数据类型的集合中选择定义一个简单类型元素

示例: <xs:simpleType name="roadbikesize">

[\[html\] view plaincopyprint?](#)

```
1. <xs:restriction base="xs:string" >
2.
3. <xs:enumeration value="small" />
4.
5. <xs:enumeration value="medium" />
6.
7. <xs:enumeration value="large" />
8.
9. </xs:restriction>
10.
11. </xs:simpleType>
12.
13. <xs:simpleType name="roadbikesize" >
14.
15. <xs:restriction base="xs:string" >
16.
17. <xs:enumeration value="small" />
18.
19. <xs:enumeration value="medium" />
```

```

20.
21. <xs:enumeration value=      />
22.
23. </xs:restriction>
24.
25. </xs:simpleType>
26.
27. </xs:schema>

```

7.complexType 类型

作用：定义一个复合类型，它决定了一组元素和属性值的约束和相关信息

•属性：name

•示例：

[\[html\] view plaincopyprint?](#)

```

1. <xs:complexType name=      > <xs:simpleContent>
2.
3. <xs:extension base=      >
4.
5. <xs:attribute name=      type=      /> </xs:extension>
6.
7. </xs:simpleContent>
8.
9. </xs:complexType>
10.
11. <xs:element name=      type=      />

```

simpleType 元素和 complexType 类型的区别（重要）

simpleType 类型的元素中不能包含元素或者属性。

•当需要声明一个元素的子元素和/或属性时，用 complexType;

- 当需要基于内置的基本数据类型定义一个新的数据类型时,用 simpleType。

8.simplecontent 元素

作用: 应用于 complexType, 对它的内容进行约束和扩展

9.choice 元素

作用: 允许唯一的一个元素从一个组中被选择

.属性: minOccurs/maxOccurs

10.sequence 元素

作用: 给一组元素一个特定的序列

一个完整的 schema 样例:

[html] view plaincopyprint?

```
1. <xs:schema xmlns:xs=      ://www.w3.org/2001/XMLSchema
2.
3.   targetNamespace=
4.
5.   xmlns=                    >
6.
7.   <xs:element name=          type=                                />
8.
9.   <xs:element name=          type=                                />
10.
11.  <xs:complexType name=      >
12.
13.    <xs:sequence>
14.
15.    <xs:element name=          type=                                />
16.
17.    <xs:element name=          type=                                />
18.
19.    <xs:element ref=          minOccurs=                            />
```

```

20.
21. <xs:element name=          type=          />
22.
23. </xs:sequence>
24.
25. <xs:attribute name=          type=          />
26.
27. </xs:complexType>
28.
29. <xs:complexType name=          >
30.
31. <xs:sequence>
32.
33. <xs:element name=          type=          />
34.
35. <xs:element name=          type=          />
36.
37. <xs:element name=          type=          />
38.
39. <xs:element name=          type=          />
40.
41. <xs:element name=          type=          />
42.
43. </xs:sequence>
44.
45. <xs:attribute name=          type=
46.
47. fixed=          />
48.
49. </xs:complexType>
50.
51. <xs:complexType name=          >
52.
53. <xs:sequence>
54.

```

```

55. <xs:element name=      minOccurs=      maxOccurs=      >
56.
57. <xs:complexType>
58.
59. <xs:sequence>
60.
61. <xs:element name=      type=      />
62.
63. <xs:element name=      >
64.
65. <xs:simpleType>
66.
67. <xs:restriction base=      >
68.
69. <xs:maxExclusive value=      />
70.
71. </xs:restriction>
72.
73. </xs:simpleType>
74.
75. </xs:element>
76.
77. <xs:element name=      type=      />
78.
79. <xs:element ref=      minOccurs=      />
80.
81. <xs:element name=      type=      minOccurs=      />
82.
83. </xs:sequence>
84.
85. <xs:attribute name=      type=      use=      />
86.
87. </xs:complexType>
88.
89. </xs:element>

```



```
90.
91. </xs:sequence>
92.
93. </xs:complexType>
94.
95. <!-- Stock Keeping Unit, a code for identifying products -->
96.
97. <xs:simpleType name=      >
98.
99. <xs:restriction base=      >
100.
101.   <xs:minInclusive="2"/>
102.
103.   <xs:maxInclusivexs:maxInclusive="10">
104.
105.   </xs:restriction>
106.
107. </xs:simpleType>
108.
109. </xs:schema>
```

Schema 总结:

Schema 是另一种文档类型定义，它遵循 XML 的语言规范。

.Schema 是可扩展的，支持命名空间；

.Schema 支持更多的数据类型与元素类型；

.Schema 用 element 声明元素，用 attribute 声明元素的属性；

.Schema 用 simpleType 定义简单类型，用 complexType 定义复杂类型。

（二十六）XML 之 DOM 和 SAX 解析

DOM 解析

在 DOM 接口规范中，有四个基本的接口：**Document**，**Node**，**NodeList** 以及 **NamedNodeMap**。在这四个基本接口中，**Document** 接口是对文档进行操作的入口，它是从 **Node** 接口继承过来的。**Node** 接口是其他大多数接口的父类，象 **Document**，**Element**，**Attribute**，**Text**，**Comment** 等接口都是从 **Node** 接口继承过来的。**NodeList** 接口是一个节点的集合，它包含了某个节点中的所有子节点。**NamedNodeMap** 接口也是一个节点的集合，通过该接口，可以建立节点名和节点之间的一一映射关系，从而利用节点名可以直接访问特定的节点。

1.Document

Document 接口代表了整个 XML/HTML 文档，因此，它是整棵文档树的根，提供了对文档中的数据进行访问和操作的入口。

2.NodeList

NodeList 接口提供了对节点集合的抽象定义，它并不包含如何实现这个节点集的定义。**NodeList** 用于表示有顺序关系的一组节点，比如某个节点的子节点序列。另外，它还出现在一些方法的返回值中，例如 **getElementsByTagName**。

在 DOM 中，**NodeList** 的对象是"live"的，换句话说，对文档的改变，会直接反映到相关的 **NodeList** 对象中。例如，如果通过 DOM 获得一个 **NodeList** 对象，该对象中包含了某个 **Element** 节点的所有子节点的集合，那么，当再通过 DOM 对 **Element** 节点进行操作（添加、删除、改动节点中的子节点）

时，这些改变将会自动地反映到 **NodeList** 对象中，而不需 **DOM** 应用程序再做其他额外的操作。

NodeList 中的每个 **item** 都可以通过一个索引来访问，该索引值从 0 开始。

3.NamedNodeMap

实现了 **NamedNodeMap** 接口的对象中包含了可以通过名字来访问的一组节点的集合。不过注意，**NamedNodeMap** 并不是从 **NodeList** 继承过来的，它所包含的节点集中的节点是无序的。尽管这些节点也可以通过索引来进行访问，但这只是提供了枚举 **NamedNodeMap** 中所包含节点的一种简单方法，并不表明在 **DOM** 规范中为 **NamedNodeMap** 中的节点规定了一种排列顺序。

NamedNodeMap 表示的是一组节点和其唯一名字的一一对应关系，这个接口主要用在属性节点的表示上。与 **NodeList** 相同，在 **DOM** 中，**NamedNodeMap** 对象也是"live"的。

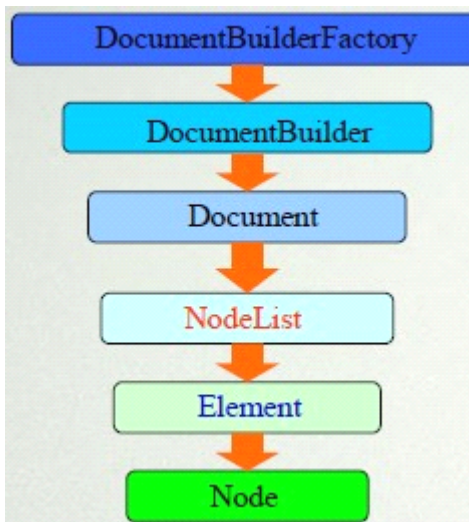
4.Dom 对象

一切都是节点（对象）

.Node 对象：DOM 结构中最基本的对象

- Document 对象：代表整个 XML 的文档
- NodeList 对象：包含一个或者多个 Node 的列表
- Element 对象：代表 XML 文档中的标签元素

5.dom 解析 xml 步骤



[java] [view plaincopyprint?](#)

```
1. import javax.xml.parsers.*;
2.
3. import org.w3c.dom.*;
4.
5. public class dom {
6.
7.     public static void main(String args[]){
8.
9.         try{
10.
11.             //建立解析器工厂
12.
13.             DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
14.
15.             //获得解析器
16.
17.             DocumentBuilder builder=factory.newDocumentBuilder();
18.
19.             Document doc=builder.parse("candidate.xml");
20.
21.             NodeList nl =doc.getElementsByTagName("PERSON");
22.
23.             for (int i=0;i<nl.getLength();i++){
```

```

24.
25.Element node=(Element) nl.item(i);
26.
27.System.out.print("NAME: ");
28.
29.System.out.println (node.getElementsByTagName("NAME").item(0).getFirstChild().
    getNodeValue());
30.
31.....
32.
33.System.out.println();
34.
35.}
36.
37.}catch(Exception e){e.printStackTrace();}
38.
39.}
40.
41.}

```

程序详解:

1)DocumentBuilderFactory dbf=DocumentBuilderFactory.newInstance();

•我们在这里使用 DocumentBuilderFactory 的目的是为了创建与具体解析器无关的程序，当 DocumentBuilderFactory 类的静态方法 newInstance()被调用时，它根据一个系统变量来决定具体使用哪一个解析器。又因为所有的解析器都服从于 JAXP 所定义的接口，所以无论具体使用哪一个解析器，代码都是一样的。所以当在不同的解析器之间进行切换时，只需要更改系统变量的值，而不用更改任何代码。这就是工厂所带来的好处。

2) •DocumentBuilder db = dbf.newDocumentBuilder();

- 当获得一个工厂对象后，使用它的静态方法 `newDocumentBuilder()`方法可以获得一个 `DocumentBuilder` 对象，这个对象代表了具体的 DOM 解析器。

但具体是哪一种解析器，微软的或者 IBM 的，对于程序而言并不重要

3)然后，我们就可以利用这个解析器来对 XML 文档进行解析了

- `Document doc = db.parse("c:/xml/message.xml");`

- `DocumentBuilder` 的 `parse()`方法接受一个 XML 文档名作为输入参数，返回一个 `Document` 对象，这个 `Document` 对象就代表了一个 XML 文档的树模型。以后所有的对 XML 文档的操作，都与解析器无关，直接在这个 `Document` 对象上进行操作就可以了。而具体对 `Document` 操作的方法，就是由 DOM 所定义的了

4) .从上面得到的 `Document` 对象开始，我们就可以开始我们的 DOM 解析了。使用 `Document` 对象的 `getElementsByTagName()`方法，我们可以得到一个 `NodeList` 对象，一个 `Node` 对象代表了一个 XML 文档中的一个标签元素，而 `NodeList` 对象，所代表的是一个 `Node` 对象的列表

`NodeList nl = doc.getElementsByTagName("message");` •我们通过这样一条语句所得到的是 XML 文档中所有<message>标签对应的 `Node` 对象的一个列表。然后，我们可以使用 `NodeList` 对象的 `item()`方法来得到列表中的每一个 `Node` 对象

- `Node my_node = nl.item(0);`

5) .当一个 `Node` 对象被建立之后，保存在 XML 文档中的数据就被提取出来并封装在这个 `Node` 中了。在这个例子中，要提取 `Message` 标签内的内容，我们通常会使用 `Node` 对象的 `getNodeValue()`方法

- `Stringmessage`

```
=my_node.getFirstChild().getNodeValue();
```

注意： 请注意，这里还使用了一个 `getFirstChild()` 方法来获得 `message` 下面的第一个子 `Node` 对象。虽然在 `message` 标签下面除了文本外并没有其它子标签或者属性，但是我们坚持在这里使用 `getFirstChild()` 方法，这主要和 W3C 对 DOM 的定义有关。W3C 把标签内的文本部分也定义成一个 `Node`，所以先要得到代表文本的那个 `Node`，我们才能够使用 `getNodeValue()` 来获取文本的内容

6.dom 基本对象详解

DOM 的基本对象有 5 个：Document，Node，NodeList，Element 和 Attr

Document 对象代表了整个 XML 的文档，所有其它的 `Node`，都以一定的顺序包含在 `Document` 对象之内，排列成一个树形的结构，程序员可以通过遍历这颗树来得到 XML 文档的所有内容，这也是对 XML 文档操作的起点。我们总是先通过解析 XML 源文件而得到一个 `Document` 对象，然后再来执行后续的操作。此外，`Document` 还包含了创建其它节点的方法，比如 `createAttribute()` 用来创建一个 `Attr` 对象。它所包含的主要的方法有

1) **.createAttribute(String):** 用给定的属性名创建一个 `Attr` 对象，并可在其后使用 `setAttributeNode` 方法来放置在某一个 `Element` 对象上面。

•2) **createElement(String):** 用给定的标签名创建一个 `Element` 对象，代表 XML 文档中的一个标签，然后就可以在这个 `Element` 对象上添加属性或进行其它的操作。

•3) **createTextNode(String):** 用给定的字符串创建一个 `Text` 对象，`Text` 对象代表了标签或者属性中所包含的纯文本字符串。如果在一个标签内没有其

它的标签，那么标签内的文本所代表的 **Text** 对象是这个 **Element** 对象的唯一子对象。

4)getElementsByTagName(String): 返回一个 **NodeList** 对象，它包含了所有给定标签名字的标签。

5)getDocumentElement(): 返回一个代表这个 **DOM** 树的根元素节点的 **Element** 对象，也就是代表 **XML** 文档根元素的那个对象。

7.Node 对象所包含的主要的方法有

- appendChild(org.w3c.dom.Node):** 为这个节点添加一个子节点，并放在所有子节点的最后，如果这个子节点已经存在，则先把它删掉再添加进去。

- getFirstChild():** 如果节点存在子节点，则返回第一个子节点，对等的，还有 **getLastChild()**方法返回最后一个子节点。

- getNextSibling():** 返回在 **DOM** 树中这个节点的下一个兄弟节点，对等的，还有 **getPreviousSibling()**方法返回其前一个兄弟节点。

- getNodeName():** 根据节点的类型返回节点的名称。

- getNodeType():** 返回节点的类型

- getNodeValue():** 返回节点的值。

- hasChildNodes():** 判断是不是存在有子节点。

- hasAttributes():** 判断这个节点是否存在有属性。

- getOwnerDocument():** 返回节点所处的 **Document** 对象。

- insertBefore(org.w3c.dom.Node new, org.w3c.dom.Node ref):** 在给定的一个子对象前再插入一个子对象。

- removeChild(org.w3c.dom.Node):** 删除给定的子节点对象

`.replaceChild(org.w3c.dom.Node new, org.w3c.dom.Node old)`: 用一个新的 **Node** 对象代替给定的子节点对象。

•**NodeList** 对象，顾名思义，就是代表了一个包含了一个或者多个 **Node** 的列表。可以简单的把它看成一个 **Node** 的数组，我们可以通过方法来获得列表中的元素：

•**getLength()**: 返回列表的长度。

•**item(int)**: 返回指定位置的 **Node** 对象

8 Element 对象代表的是 XML 文档中的标签元素，继承于 Node，亦是 **Node** 的最主要的子对象。在标签中可以包含有属性，因而 **Element** 对象中有存取其属性的方法，而任何 **Node** 中定义的方法，也可以用在 **Element** 对象上面。

•**getElementsByTagName(String)**: 返回一个 **NodeList** 对象，它包含了在这个标签中其下的子孙节点中具有给定标签名字的标签。

•**getTagName()**: 返回一个代表这个标签名字的字符串。

•**getAttribute(String)**: 返回标签中给定属性名称的属性的值。在这儿需要注意的是，因为 XML 文档中允许有实体属性出现，而这个方法对这些实体属性并不适用。这时候需要用到 **getAttributeNode()**方法来得到一个 **Attr** 对象来进行进一步的操作

•**getAttributeNode(String)**: 返回一个代表给定属性名称的 **Attr** 对象。

9. Attr 对象代表了某个标签中的属性。Attr 继承于 **Node**，但是因为 Attr 实际上是包含在 **Element** 中的，它并不能被看作是 **Element** 的子对象，因而在 DOM 中 Attr 并不是 DOM 树的一部分，所以 **Node** 中的 **getParentNode()**，**getPreviousSibling()**和 **getNextSibling()**返回的都将是 **null**。也就是说，Attr

其实是被看作包含它的 **Element** 对象的一部分，它并不作为 **DOM** 树中单独的一个节点出现。这一点在使用的时候要同其它的 **Node** 子对象相区别

SAX 解析

SAX 的全称是 **Simple APIs for XML**，**SAX** 提供的访问模式是一种顺序模式，这是一种快速读写 **XML** 数据的方式。当使用 **SAX** 分析器对 **XML** 文档进行分析时，会触发一系列事件，并激活相应的事件处理函数，应用程序通过这些事件处理函数实现对 **XML** 文档的访问，因而 **SAX** 接口也被称作事件驱动接口。由于 **SAX** 分析器实现简单，对内存要求比较低，因此实现效率比较高，对于那些只需要访问 **XML** 文档中的数据而不对文档进行更改的应用程序来说，**SAX** 分析器更为合适。

SAX (**Simple APIs for XML**)，面向 **XML** 的简单 **APIs**。使用 **DOM** 解析 **XML** 时，首先将 **XML** 文档加载到内存当中，然后通过随机的方式访问内存中的 **DOM** 树；**SAX** 是基于事件而且是顺序执行的，一旦经过了某个元素，我们就没有办法再去访问它了，**SAX** 不必事先将整个 **XML** 文档加载到内存当中，因此它占据内存要比 **DOM** 小，对于大型的 **XML** 文档来说，通常会使用 **SAX** 而不是 **DOM** 进行解析。

SAX 也是使用的观察者模式（类似于 **GUI** 中的事件）

SAX分析器的大体构成框架



图中最上方的 `SAXParserFactory` 用来生成一个分析器实例。XML 文档是从左侧箭头所示处读入，当分析器对文档进行分析时，就会触发在 `DocumentHandler`，`ErrorHandler`，`DTDHandler` 以及 `EntityResolver` 接口中定义的回调方法。

4.SAX 是事件驱动的，文档的读入过程就是 SAX 的解析过程。在读入的过程中，遇到不同的项目，解析器会调用不同的处理方法。

5.org.xml.sax.helpers.DefaultHandler 类的方法

项目	处理方法
文档开始	<code>startDocument()</code>
<code><PEOPLE></code>	<code>startElement()</code>
<code>"Tony Blair"</code>	<code>characters()</code>
<code></PEOPLE></code>	<code>endElement()</code>
文档结束	<code>endDocument()</code>

6.SAX 方式提取 XML 文档内容信息示例

[java] [view plaincopyprint?](#)

```
1. package com.shengsiyuan.xml.sax;
2.
3. import java.io.File;
4.
5. import java.util.Stack;
6.
7. import javax.xml.parsers.SAXParser;
8.
9. import javax.xml.parsers.SAXParserFactory;
10.
11. import org.xml.sax.Attributes;
12.
13. import org.xml.sax.SAXException;
14.
15. import org.xml.sax.helpers.DefaultHandler;
16.
17. public class SaxTest2
18. {
19. {
20.
21. public static void main(String[] args) throws Exception
22. {
23. {
24.
25.         //step1: 获得 SAX 解析器工厂实例
26.
27. SAXParserFactory factory = SAXParserFactory.newInstance();
28.
29.         //step2: 获得 SAX 解析器实例
30.
31. SAXParser parser = factory.newSAXParser();
32.
33.         //step3: 开始进行解析
34.
35. parser.parse(new File("student.xml"), new MyHandler2());
```

```

36.
37.}
38.
39.}
40.
41.class MyHandler2 extends DefaultHandler
42.
43.{
44.
45.private Stack<String> stack = new Stack<String>();
46.
47.private String name;
48.
49.private String gender;
50.
51.private String age;
52.
53.@Override
54.
55.public void startElement(String uri, String localName, String qName,
56.
57.Attributes attributes) throws SAXException
58.
59.{
60.
61.stack.push(qName);
62.
63.for(int i = 0; i < attributes.getLength(); i++)
64.
65.{
66.
67.String attrName = attributes.getQName(i);
68.
69.String attrValue = attributes.getValue(i);
70.

```

```
71.System.out.println(attrName + "=" + attrValue);
72.
73.}
74.
75.}
76.
77.@Override
78.
79.public void characters(char[] ch, int start, int length)
80.
81.throws SAXException
82.
83.{
84.
85.String tag = stack.peek();
86.
87.if("姓名".equals(tag))
88.
89.{
90.
91.name = new String(ch, start,length);
92.
93.}
94.
95.else if("性别".equals(tag))
96.
97.{
98.
99.gender = new String(ch, start, length);
100.
101.    }
102.
103.    else if("年龄".equals(tag))
104.
105.    {
```

```
106.
107.     age = new String(ch, start, length);
108.
109. }
110.
111. }
112.
113.     @Override
114.
115.     public void endElement(String uri, String localName, String qName)
116.
117.     throws SAXException
118.
119.     {
120.
121.         stack.pop(); //表示该元素已经解析完毕，需要从栈中弹出
122.
123.
124.         if("学生".equals(qName))
125.
126.         {
127.
128.             System.out.println("姓名: " + name);
129.
130.             System.out.println("性别: " + gender);
131.
132.             System.out.println("年龄: " + age);
133.
134.             System.out.println();
135.
136.         }
137.
138.     }
139.
140. }
```


（二十七）XML 之 Jdom 和 DOM4J 解析 .

jdome 解析

JDOME 是一种使用 XML 的独特 Java 工具包，用于快速开发 XML 应用程序。它的设计包含 Java 语言的语法乃至语义。JDOME 是一个开源项目，它基于树型结构，利用纯 JAVA 的技术对 XML 文档实现解析、生成、序列化以及多种操作。（<http://jdom.org>）

JDOME 直接为 JAVA 编程服务。它利用更为强有力的 JAVA 语言的诸多特性(方法重载、集合概念等),把 SAX 和 DOM 的功能有效地结合起来。DOM 是用 Java 语言读、写、操作 XML 的新 API 函数。在直接、简单和高效的前提下，这些 API 函数被最大限度的优化。在使用设计上尽可能地隐藏原来使用 XML 过程中的复杂性。利用 JDOME 处理 XML 文档将是一件轻松、简单的事。JDOME 主要用来弥补 DOM 及 SAX 在实际应用当中的不足之处。这些不足之处主要在于 SAX 没有文档修改、随机访问以及输出的功能，而对于 DOM 来说，JAVA 程序员在使用时来用起来总觉得不太方便。DOM 的缺点主要是由于 DOM 是一个接口定义语言（IDL）,它的任务是在不同语言实现中的一个最低的通用标准，并不是为 JAVA 特别设计的。在 JDOME 中，XML 元素就是 Element 的实例，XML 属性就是 Attribute 的实例，XML 文档本身就是 Document 的实例.JDOME 是作为一种轻量级 API 被制定的，最主要的是它是以 Java 为中心的。它在遵循 DOM 主要规则的基础上除去了 dom 本身的缺点。

因为 JDOM 对象就是像 Document、Element 和 Attribute 这些类的直接实例，因此创建一个新 JDOM 对象就如在 Java 语言中使用 new 操作符一样容易。JDOM 的使用是直截了当的。JDOM 使用标准的 Java 编码模式。只要有可能，它使用 Java new 操作符而不使用复杂的工厂模式，使对象操作即便对于初学用户也很方便。

JDOM 是由以下几个包组成的

- org.jdom 包含了所有的 xml 文档要素的 java 类
 - org.jdom.adapters 包含了与 dom 适配的 java 类
 - org.jdom.filter 包含了 xml 文档的过滤器类
 - org.jdom.input 包含了读取 xml 文档的类
 - org.jdom.output 包含了写入 xml 文档的类
 - org.jdom.transform 包含了将 jdomxml 文档接口转换为其他 xml 文档接口
 - org.jdom.xpath 包含了对 xml 文档 xpath 操作的类
- org.jdom 这个包里的类是你解析 xml 文件后所要用的所有数据类型。–Attribute –CDATA –Comment –DocType –Document –Element –Entity Ref –Namespace –ProcessingInstruction –Text

Jdom 主要使用方法：

1.Document 类

Document 的操作方法：

```
Element root = new Element("GREETING");
Document doc = new Document(root);
root.setText("Hello JDOM!");
```

或者简单的使用

```
Document doc=new  
Document(new Element("GREETING").setText("Hello JDOM!t"));
```

2.这点和 DOM 不同。Dom 则需要更为复杂的代码，如下：

[html] [view plaincopyprint?](#)

```
1. DocumentBuilderFactory factory=                .newInstance();  
2.  
3. DocumentBuilder builder =                .newDocumentBuilder();  
4.  
5. Document doc =                .newDocument();  
6.  
7. Element root =                .createElement("root");  
8.  
9. Text text =                .createText("This is the root");  
10.  
11.root.appendChild(text);  
12.  
13.doc.appendChild(root);
```

3.可以使用 SAXBuilder 的 build 方法来解析一个流从而得到一个 Document 对象

[html] [view plaincopyprint?](#)

```
1. -Document build(java.io.File file)  
2.  
3. -Document build(org.xml.sax.InputSource in)  
4.  
5. -Document build(java.io.InputStream in)  
6.  
7. -Document build(java.net.URL url)
```

4.DOM 的 Document 和 JDOM 的 Document 之间的相互转换使用方法

[html] [view plaincopyprint?](#)

```
1. DOMBuilder builder =                DOMBuilder();
```

```

2.
3. org.jdom.Document jdomDocument =      .build(domDocument);
4.
5. -DOMOutputter converter =      DOMOutputter();// work with the JDOM document...
6.
7. -org.w3c.dom.Document domDocument =      .output(jdomDocument);
8.
9. -// work with the DOM document...

```

5.XMLOutPutter 类:

JDOM 的输出非常灵活,支持很多种 io 格式以及风格的输出

[html] [view plaincopyprint?](#)

```

1. Document doc =      Document(...);
2.
3. XMLOutputter outp =      XMLOutputter();
4.
5. outp.output(doc, fileOutputStream); // Raw output
6.
7. outp.setTextTrim(true); // Compressed output
8.
9. outp.output(doc, socket.getOutputStream());
10.
11.outp.setIndent(" "); // Pretty output
12.
13.outp.setNewlines(true);
14.
15.outp.output(doc, System.out);

```

DOM4J 解析

DOM4J 是 dom4j.org 出品的一个开源 XML 解析包，它是一个易用的、开源的库，用于 XML，XPath 和 XSLT。它应用于 Java 平台，采用了 Java 集合框架并完全支持 DOM，SAX 和 JAXP。DOM4J 使用起来非常简单。只要你了解基本的 XML-DOM 模型，就能使用。

它的主要接口都在 org.dom4j 这个包里定义：

Attribute	Attribute 定义了 XML 的属性
Branch	Branch 为能够包含子节点的节点如 XML 元素 (Element)和文档(Docuemnts)定义了一个公共的行为，
CDATA	CDATA 定义了 XML CDATA 区域
CharacterData	CharacterData 是一个标识借口，标识基于字符的节点。如 CDATA, Comment, Text.
Comment	Comment 定义了 XML 注释的行为
Document	定义了 XML 文档
DocumentType	DocumentType 定义 XML DOCTYPE 声明
Element	Element 定义 XML 元素
ElementHandler	ElementHandler 定义了 Element 对象的处理器
ElementPath	被 ElementHandler 使用，用于取得当前正在处理的路径层次信息
Entity	Entity 定义 XML entity
Node	Node 为所有的 dom4j 中 XML 节点定义了多态行为
NodeFilter	NodeFilter 定义了 dom4j 节点中产生的一个滤镜或谓词的行为 (predicate)
ProcessingInstruction	ProcessingInstruction 定义 XML 处理指令。
Text	Text 定义 XML 文本节点。
Visitor	Visitor 用于实现 Visitor 模式。
XPath	XPath 在分析一个字符串后会提供一个 XPath 表达式

看名字大致就知道它们的涵义如何了。下面咱一一看一下：

一.Document 对象,三种创建方法

1.读取 XML 文件,获得 document 对象.

```
SAXReader reader = new SAXReader();
Document document = reader.read(new File("input.xml"));
```

2.解析 XML 格式的字符串,获得 document 对象.

```
String text = "<members></members>";  
Document document = DocumentHelper.parseText(text);
```

3.创建 document 空对象.

```
Document document = DocumentHelper.createDocument();  
Element root = document.addElement("members");// 创建根节点,
```

只有空 DOCUMENT 对象才能创建 ROOT 结点

二.节点控制

1.获取文档的根节点.

```
Element root = document.getRootElement();
```

2.取得节点的文本

```
String text=memberElm.getText();
```

也可以用:

```
String text=root.elementText("name"); //这个是取得根节点下的
```

name 字节点的文字;可以类推任何节点下的文本

3.设置节点文字.

```
ageElm.setText("29");
```

4.父节点下获得单个子节点对象.

```
Element memberElm=root.element("member"); // "member"是节点
```

名

5.取得父节点下遍历名为"member"的所有子节点.

```
List nodes = rootElm.elements("member");  
for (Iterator it = nodes.iterator(); it.hasNext();) {  
    Element elm = (Element) it.next();  
    // do something
```

```
}
```

6.父节点下的遍历所有子节点进行.

```
for(Iterator it=root.elementIterator();it.hasNext();){  
    Element element = (Element) it.next();  
    // do something  
}
```

7.父节点下添加子节点.

```
Element ageElm = newMemberElm.addElement("age");
```

8.父节点下删除子节点.

```
parentElm.remove(childElm);// childElm 是待删除的节
```

点,parentElm 是其父节点

三.属性相关.

1.取得某节点下的某属性

```
Element root=document.getRootElement();  
Attribute attribute=root.attribute("size");// 属性名 name
```

2.取得属性的文字

```
String text=attribute.getText();
```

也可以用:

```
String text2=root.element("name").attributeValue("firstname");这个是取得  
根节点下 name 字节点的属性 firstname 的值.
```

3.遍历某节点的所有属性

```
Element root=document.getRootElement();  
for(Iterator it=root.attributeIterator();it.hasNext();){  
    Attribute attribute = (Attribute) it.next();  
    String text=attribute.getText();
```

```
        System.out.println(text);
    }
```

4.设置某节点的属性和文字.

```
newMemberElm.addAttribute("name", "sitinspring");
```

5.设置属性的文字

```
Attribute attribute=root.attribute("name");
attribute.setText("sitinspring");
```

6.删除某属性

```
Attribute attribute=root.attribute("size");// 属性名 name
root.remove(attribute);
```

四.将文档写入 **XML** 文件.

1.文档中全为英文,不设置编码,直接写入的形式.

```
XMLWriter writer = new XMLWriter(new FileWriter("output.xml"));
writer.write(document);
writer.close();
```

2.文档中含有中文,设置编码格式写入的形式.

```
OutputFormat format = OutputFormat.createPrettyPrint();
format.setEncoding("GBK"); // 指定 XML 编码

XMLWriter writer = new XMLWriter(new FileWriter("output.xml"),format);
writer.write(document);
writer.close();
```

五.字符串与 **XML** 的转换

1.将字符串转化为 XML

```
String text = "<members> <member>sitinspring</member> </members>";
Document document = DocumentHelper.parseText(text);
```

2.将文档或节点的 XML 转化为字符串.

```
SAXReader reader = new SAXReader();
Document document = reader.read(new File("input.xml"));
```



```
Element root=document.getRootElement();
String docXmlText=document.asXML();
String rootXmlText=root.asXML();
Element memberElm=root.element("member");
String memberXmlText=memberElm.asXML();
```

六.使用 XPath 快速找到节点.

读取的 XML 文档示例

[html] [view plaincopyprint?](#)

```
1. <?xml version=      encoding=      ?>
2. <projectDescription>
3.   <name>MemberManagement</name>
4.   <comment></comment>
5.   <projects>
6.     <project>PRJ1</project>
7.     <project>PRJ2</project>
8.     <project>PRJ3</project>
9.     <project>PRJ4</project>
10.  </projects>
11.  <buildSpec>
12.    <buildCommand>
13.      <name>org.eclipse.jdt.core.javabuilder</name>
14.      <arguments>
15.        </arguments>
16.    </buildCommand>
17.  </buildSpec>
18.  <natures>
19.    <nature>org.eclipse.jdt.core.javanature</nature>
20.  </natures>
21.</projectDescription>
```

使用 XPath 快速找到节点 project.

[html] [view plaincopyprint?](#)

```
1. public static void main(String[] args){
2.     SAXReader reader =      SAXReader();
3.     try{
4.         Document doc =      .read(new File("sample.xml"));
5.         List projects=      .selectNodes("/projectDescription/projects/project");
6.         //Element nodes0=      .selectSingleNode("/bookstore"); //采用相对路径,即
           当前结点(包括当前结点)开始查找,与下列结果相同.
7.         //Element nodes=      .selectNodes("book");      ////采用绝对路径,即当前结点
           (包括当前结点)开始查找,
8.         //XPATH 语法详见: http://www.w3school.com.cn/xpath/xpath\_syntax.asp
9.         Iterator it=      .iterator();
10.        while(it.hasNext()){
11.            Element elm=(Element)it.next();
12.            System.out.println(elm.getText());
13.        }
14.    }
15.    catch(Exception ex){
16.        ex.printStackTrace();
17.    }
18. }
```

（二十八）Javascript 总结之语言基础

JavaScript 脚本语言作为一门功能强大、使用范围较广的程序语言，其语言基础包括数据类型、变量、运算符、函数以及核心语句等内容。本篇文章主要介绍 **JavaScript** 脚本语言的基础知识

一：基础常识

1.脚本执行顺序：**JavaScript** 脚本解释器将按照程序代码出现的顺序来解释程序语句，因此可以将函数定义和变量声明放在<head>和</head>之间，此时与函数体相关的操作不会被立即执行。

2.大小写敏感：**JavaScript** 脚本程序对大小写敏感，相同的字母，大小写不同，代表的意义也不同

空白字符

3.空_____白字符包括空格、制表符和换行符等，在编写脚本代码时占据一定的空间，但脚本被浏览器解释执行时无任何作用。脚本程序员经常使用空格作为空白字符，**JavaScript** 脚本解释器是忽略任何多余空格的

注：在字符串中，空格不被忽略，而作为字符串的一部分显示出来，在编写 **JavaScript** 脚本代码时，经常需添加适当的空格使脚本代码层次明晰，方便相关人员查看和维护。

4.分号：在编写脚本语句时，用分号作为当前语句的结束符

5.块：在定义函数时，使用大括号“{}”将函数体封装起来

二：数据类型

首先要说明的是在 **javascript** 中没有特定的数据类型修饰符，基本数据类型不管是什么类型，在定义的时候都是用 **var**

1.整型和浮点数值：JavaScript 允许使用整数类型和浮点类型两种数值，其中整数类型包含正整数、0 和负整数；而浮点数则可以是包含小数点的实数，也可以是用科学计数法表示的实数

2 八进制和十六进制：在整数类型的数值中，数制可使用十进制、八进制以及十六进制

三. 变量

1.变量标识符：与 C++、Java 等高级程序语言使用多个变量标识符不同，JavaScript 脚本语言使用关键字 **var** 作为其唯一的变量标识符，其用法为在关键字 **var** 后面加上变量名。

2.变量作用域：要讨论变量的作用域，首先要清楚全局变量和局部变量的联系和区别：

◆◆◆ **全局变量：**可以在脚本中的任何位置被调用，全局变量的作用域是当前文档中整个脚本区域。

◆◆◆ **局部变量：**只能在此变量声明语句所属的函数内部使用，局部变量的作用域仅为该函数体。

声明变量时，要根据编程的目的决定将变量声明为全局变量还是局部变量。

一般而言，保存全局信息（如表格的原始大小、下拉框包含选项对应的字符串数组等）的变量需声明为全局变量，而保存临时信息（如待输出的格式字符串、数学运算中间变量等）的变量则声明为局部变量。

3.弱类型：JavaScript 脚本语言像其他程序语言一样，其变量都有数据类型，具体数据类型将在下一节中介绍。高级程序语言如 C++、Java 等为强类型语言，与此不同的是，JavaScript 脚本语言是弱类型语言，在变量声明时不需显式地指定其数据类型，变量的数据类型将根据变量的具体内容推导出来，且根据变量内容的改变而自动更改，而强类型语在变量声明时必须显式地指定其数据类型。变量声明时不需显式指定其数据类型既是 JavaScript 脚本语言的优点也是缺点，优点是编写脚本代码时不需要指明数据类型，使变量声明过程简单明了；缺点就是有可能造成因微妙的拼写不当而引起致命的错误。

4.基本数据类型：在实现预定功能的程序代码中，一般需定义变量来存储数据（作为初始值、中间值、最终值或函数参数等）。变量包含多种类型，JavaScript 脚本语言支持的基本数据类型包括

Number 型、String 型、Boolean 型、Undefined 型、Null 型和 Function 型，下面简单介绍一下几种数据类型：

4.1 Number 型：Number 型数据即为数值型数据，包括整数型和浮点型，整数型数制可以使用十进制、八进制以及十六进制标识，而浮点型为包含小数点的实数，且可用科学计数法来表示。一般来说，Number 型数据为不在括号内的数字

4.2 Undefined 型：Undefined 型即为未定义类型，用于不存在或者没有被赋初始值的变量或对象的属性，如下列语句定义变量 name 为 Undefined 型：
`var name;`定义 Undefined 型变量后，可在后续的脚本代码中对其进行赋值操作，从而自动获得由其值决定的数据类型。

4.2 Null 型：Null 型数据表示空值，作用是表明数据空缺的值，一般在设定已存在的变量（或对象的属性）为空时较为常用。区分 Undefined 型和 Null 型数据比较麻烦，一般将 Undefined 型和 Null 型等同对待。

4.3 Function 型：Function 型表示函数，可以通过 new 操作符和构造函数 Function() 来动态创建所需功能的函数，并为其添加函数体。例如：

```
var myFuntion = new Function()  
{staments};
```

JavaScript 脚本语言除了支持上述六种基本数据类型外，也支持组合类型，如数组 Array 和对象 Object 等，下面介绍组合类型。

四：组合类型

1.Array 型：Array 型即为数组，数组是包含基本和组合数据的序列。在 JavaScript 脚本语言中，每一种数据类型对应一种对象，数组本质上即为 Array 对象。考察如下定义：

```
var score = [56,34,23,76,45];
```

上述语句创建数组 score，中括号“[]”内的成员为数组元素。由于

JavaScript 是弱类型语言，因此不要求目标数组中各元素的数据类型均相同，

例如：var score = [56,34,"23",76,"45"];

由于数组本质上为 Array 对象，则可用运算符 new 来创建新的数组，例如：

```
var score=new Array(56,34,"23",76,"45");
```

访问数组中特定元素可通过该元素的索引位置 index 来实现，如下列语句声明变量 m 返回数组 score 中第四个元素：

```
var m = score [3];
```

数组作为 Array 对象，具有最重要的属性 length，用来保存该数组的长度

2 Object 型：对象为可包含基本和组合数据的组合类型，且对象的成员作为对象的属性，对象的成员函数作为对象的方法。在 JavaScript 脚本语言中，可通过在对象后面加句点“.”并加上对象

属性（或方法）的名称来访问对象的属性（或方法），例如：

```
document.bgColor
```

```
document.write("Welcome to JavaScript World!");
```

五：运算符

1 赋值运算符：JavaScript 脚本语言的赋值运算符包含“=”、“+=”、“-=”、“*=”、“/=”、“%=”、“&=”、“^=”等

运算符 举例 简要说明

= m=n 将运算符右边变量的值赋给左边变量

+= m+=n 将运算符两侧变量的值相加并将结果赋给左边变量

-= m-=n 将运算符两侧变量的值相减并将结果赋给左边变量

= m=n 将运算符两侧变量的值相乘并将结果赋给左边变量

/= m/=n 将运算符两侧变量的值相除并将整除的结果赋给左边变量

%= m%=n 将运算符两侧变量的值相除并将余数赋给左边变量

&= m&=n 将运算符两侧变量的值进行按位与操作并将结果赋值给左边变量

^= m^=n 将运算符两侧变量的值进行按位或操作并将结果赋值给左边变量

<<= m<<=n 将运算符左边变量的值左移由右边变量的值指定的位数，并将操作的结果赋予左边变量

>>= `m>>=n` 将运算符左边变量的值右移由右边变量的值指定的位数，并将操作的结果赋予左边变量

>>>= `m>>>=n` 将运算符左边变量的值逻辑右移由右边变量的值指定的位数，并将操作的结果赋给左边变量

赋值运算符是编写 JavaScript 脚本代码时最为常用的操作，读者应熟练掌握各个运算符的功能，避免混淆其具体作用。

2 基本数学运算符：JavaScript 脚本语言中基本的数学运算包括加、减、乘、除以及取余等，其对应的数学运算符分别为“+”、“-”、“*”、“/”和“%”

3.位运算符：JavaScript 脚本语言支持的基本位运算符包括：“&”、“|”、“^”和“~”等。脚本代码执行位运算时先将操作数转换为二进制数，操作完成后将返回值转换为十进制

位运算符 举例 简要说明

& 9&4 按位与，若两数据对应位都是 1，则该位为 1，否则为 0

^ 9^4 按位异或，若两数据对应位相反，则该位为 1，否则为 0

| 9|4 按位或，若两数据对应位都是 0，则该位为 0，否则为 1

~ ~4 按位非，若数据对应位为 0，则该位为 1，否则为 0

位运算符在进行数据处理、逻辑判断等方面使用较为广泛，恰当应用位运算符，可节省大量脚本代码。

3.自加和自减：自加运算符为“++”和自减运算符为“--”分别将操作数加 1 或减 1。值得注意的是，

自加和自减运算符放置在操作数的前面和后面含义不同。运算符写在变量名前面，则返回值

为自加或自减前的值；而写在后面，则返回值为自加或自减后的值。

4.比较运算符：JavaScript 脚本语言中用于比较两个数据的运算符称为比较运算符，包括“= ”、“!= ”、“>”、“<”、“<= ”、“>= ”等

5.逻辑运算符：JavaScript 脚本语言的逻辑运算符包括“&&”、“||”和“!”等，用于两个逻辑型数据之间的操作，返回值的数据类型为布尔型

6.逗号运算符：编写 JavaScript 脚本代码时，可使用逗号“,”将多个语句连在一起，浏览器载入该代码时，将其作为一个完整的语句来调用，但语句的返回值是最右边的语句。逗号“,”一般用于在函数定义和调用时分隔多个参数，

7.对象运算符：JavaScript 脚本语言主要支持四种对象运算符，包括点号运算符、new 运算符、delete 运算符以及（）运算符等。对象包含属性和方法，点号运算符用来访问对象的属性和方法。其用法是将对象名称与对象的属性（或方法）用点号隔开，例如：

```
var myColor=document.bgColor;  
window.alert(msg);
```

当然，也可使用双引号“[]”来访问对象的属性，改写上述

语句：`var myColor=document[" bgColor "];`

new 运算符用来创建新的对象，例如创建一个新的数组对象，可以写成：

```
var exam = new Array (43,76,34 89,90);
```

new 运算符可以创建程序员自定义的对象，以可以创建 JavaScript 内建对象的实例。

8.typeof 运算符：typeof 运算符用于表明操作数的数据类型，返回数值类型为一个字符串。在 JavaScript 脚本语言中，其使用格式如下：

```
var myString=typeof(data);
```

六：javascript 的循环控制结构：由于在 javascript 中 if、if.....else、while、do.....while 等语句与 java 中基本相识，所以在此就不赘述

七：函数

JavaScript 脚本语言允许开发者通过编写函数的方式组合一些可重复使用的脚本代码块，增加了脚本代码的结构化和模块化。函数是通过参数接口进行数据传递，以实现特定的功能。

1 函数的基本组成

函数由函数定义和函数调用两部分组成，应首先定义函数，然后再进行调用，以养成良好的编程习惯。函数的定义应使用关键字 **function**，其语法规则如下：

```
function funcName ([parameters])  
{statements;  
[return 表达式;]}
```

除了自定义函数外，JavaScript 脚本语言提供大量的内建函数，无需开发者定义即可直接调用，例如 window 对象的 **alert()** 方法即为 JavaScript 脚本语言支持的内建函数。函数定义过程结束后，可在文档中任意位置调用该函数。引用目标函数时，只需在函数名后加上小括号即可。若目标函数需引入参数，则需在小括号内添加传递参数。如果函数有返回值，可将最终结果赋值给一个自定义的变量并用关键字 **return** 返回。

2 全局函数与局部函数：JavaScript 脚本语言提供了很多全局（内建）函数，在脚本编程过程中可直接调用，在此介绍四种简单的全局函数：`parseInt()`、`parseFloat()`、`escape()`和 `unescape()`。

`parseInt()`函数的作用是将字符串转换为整数，`parseFloat()`函数的作用是将字符串转换为浮点数；`escape()`函数的作用是将一些特殊字符转换成 ASCII 码，而 `unescape()`函数的作用是将 ASCII 码转换成字符。

3.作为对象的函数：JavaScript 脚本语言中所有的数据类型、数组等均可作为对象对待，函数也不例外。可以使用 `new` 操作符和 `Function` 对象的构造函数 `Function()`来生成指定规则的函数，其基本语法如下：

```
var funcName = new Function (arguments,statements);
```

值得注意的是，上述的构造函数 `Function()`首字母必须为大写，同时函数的参数列表与操作代码之间使用逗号隔开。

注意：在定义函数对象时，参数列表可以为空，也可有一个或多个参数，使用变量引用该函数时，应将函数执行所需要的参数传递给函数体。作为对象的函数最重要的性质即为它可以创建静态变量，给函数增加实例属性，使得函数在被调用之间也能发挥作用。作为对象的函数使用静态变量后，可以用来保存其运行的环境参数如中间值等数据。

4.函数应用注意事项

最后介绍一下在使用函数过程中应特别予以注意的几个问题，以帮助读者更好、更准确地使用函数，并养成良好的编程习惯。具体表现在如下几点：

定义函数的位置：如果函数代码较为复杂，函数之间相互调用较多，应将所有函数的定义部分放在 HTML 文档的<head>和</head>标记对之间，既可保证所有的函数在调用之前均已定义，又可使开发者后期的维护工作更为简便；

◆◆◆ 函数的命名：函数的命名原则与变量的命名原则相同，但尽量不要将函数和变量取同一个名字。如因实际情况需要将函数和变量定义相近的名字，也应给函数加上可以清楚辨认的字符（如前缀 **func** 等）以示区别；

◆◆◆ 函数返回值：在函数定义代码结束时，应使用 **return** 语句返回，即使函数不需要返回任何值；

◆◆◆ 变量的作用域：区分函数中使用的变量是全局变量还是局部变量，避免调用过程中出现难以检查的错误；

◆◆◆ 函数注释：在编写脚本代码时，应在适当的地方给代码的特定行添加注释语句，例如将函数的参数数量、数据类型、返回值、功能等注释清楚，既方便开发者对程序的后期维护，也方便其他人阅读和使用该函数，便于模块化编程；

◆◆◆ 函数参数传递：由于 JavaScript 是弱类型语言，使用变量时并不检查其数据类型，导致一个潜在的威胁，即开发者调用函数时，传递给函数的参数数量或数据类型不满足要求而导致错误的出现。在函数调用时，应仔细检查传递给目标函数的参数变量的数量和数据类型。

其中第五点尤为值得特别关注，因由其导致的错误非常难于检测。

分享到：

• 上一篇：[几种数据存储结构详解](#)

(二十九) javascript 对象的创建和继承实现

javascript 对象的创建

JavaScript 中定义对象的几种方式(JavaScript 中没有类的概念, 只有对象):

1) 基于已有对象扩充其属性和方法:

[\[html\] view plaincopyprint?](#)

```
1. var object =      Object();
2.
3. object.name =      ;
4. object.sayName =      (name)
5. {
6.   this.name =      ;
7.   alert(this.name);
8. }
9.
10.object.sayName("lisi");
```

2) 工厂方式

[\[html\] view plaincopyprint?](#)

```
1. //工厂方式创建对象
2.
3. /*
4. function createObject()
5. {
6.   var object =      Object();
7.
8.   object.username =      ;
9.   object.password =      ;
```

```

10.
11.  object.get =      ()
12.  {
13.      alert(this.username + ", " + this.password);
14.  }
15.
16.  return object;
17.}
18.
19.var object1 =      ();
20.var object2 =      ();
21.
22.object1.get();

```

带参数的构造方法:

[html] [view plain](#)[copy](#)[print?](#)

```

1. function createObject(username, password)
2. {
3.     var object =      Object();
4.
5.     object.username =      ;
6.     object.password =      ;
7.
8.     object.get =      ()
9.     {
10.         alert(this.username + ", " + this.password);
11.     }
12.
13.     return object;
14.}
15.
16.var object1 =      ("zhangsan", "123");
17.object1.get();

```

让一个函数对象被多个对象所共享，而不是每一个对象拥有一个函数对象。

[html] view plaincopyprint?

```
1. function get()
2. {
3.     alert(this.username + ", " + this.password);
4. }
5.
6. function createObject(username, password)
7. {
8.     var object =     Object();
9.
10.    object.username =     ;
11.    object.password =     ;
12.
13.    object.get =     ;
14.
15.    return object;
16.}
17.
18.var object =     ("zhangsan", "123");
19.var object2 =     ("lisi", "456");
20.
21.object.get();
22.object2.get();
```

3) 构造函数方式

[html] view plaincopyprint?

```
1. function Person()
2. {
```



```

3. //在执行第一行代码前，js 引擎会为我们生成一个对象
4. this.username =          ;
5. this.password =          ;
6.
7. this.getInfo =          ()
8. {
9.   alert(this.username + ", " + this.password);
10. }
11.
12. //此处有一个隐藏的 return 语句，用于将之前生成的对象返回
13.}
14.
15.var person =      Person();
16.person.getInfo();

```

可以在构造对象时传递参数

[html] [view plaincopyprint?](#)

```

1. function Person(username, password)
2. {
3.   this.username =          ;
4.   this.password =          ;
5.
6.   this.getInfo =          ()
7.   {
8.     alert(this.username + ", " + this.password);
9.   }
10.}
11.
12.var person =      Person("zhangsan", "123");
13.person.getInfo();

```

4) 原型 (“**prototype**”) 方式

[html] [view plaincopyprint?](#)

```
1. //使用原型（prototype）方式创建对象
2.
3. /*
4.  function Person()
5.  {
6.
7.  }
8.
9.  Person.prototype.username =      ;
10. Person.prototype.password =      ;
11.
12. Person.prototype.getInfo =      ()
13. {
14.     alert(this.username + ", " + this.password);
15. }
16.
17. var person =      Person();
18. var person2 =      Person();
19.
20. person.username =      ;
21.
22. person.getInfo();
23. person2.getInfo();
24. */
```

[html] [view plaincopyprint?](#)

```
1.  function Person()
2.  {
3.
4.  }
5.
```

```

6. Person.prototype.username = Array();
7. Person.prototype.password = ;
8.
9. Person.prototype.getInfo = ()
10.{
11.  alert(this.username + ", " + this.password);
12.}
13.
14.var person = Person();
15.var person2 = Person();
16.
17.person.username.push("zhangsan");
18.person.username.push("lisi");
19.person.password = ;
20.
21.person.getInfo();
22.person2.getInfo();

```

如果使用原型方式对象，那么生成的所有对象会共享原型中的属性，这样一个对象改变了该属性也会反应到其他对象当中。

单纯使用原型方式定义对象无法在构造函数中为属性赋初值，只能在对象生成后再去改变属性值。

使用原型+构造函数方式来定义对象，对象之间的属性互不干扰，各个对象间共享同一个方法

[html] [view plaincopyprint?](#)

```

1. //使用原型+构造函数方式来定义对象
2.
3. function Person()
4. {
5.  this.username = Array();

```

```

6.   this.password =      ;
7. }
8.
9. Person.prototype.getInfo =      ()
10. {
11.   alert(this.username + ", " + this.password);
12. }
13.
14. var p =      Person();
15. var p2 =      Person();
16.
17. p.username.push("zhangsan");
18. p2.username.push("lisi");
19.
20. p.getInfo();
21. p2.getInfo();

```

5) 动态原型方式：在构造函数中通过标志量让所有对象共享一个方法，而每个对象拥有自己的属性。

[html] [view plain](#)copyprint?

```

1. function Person()
2. {
3.   this.username =      ;
4.   this.password =      ;
5.
6.   if(typeof Person.flag == "undefined")
7.   {
8.     alert("invoked");
9.
10.    Person.prototype.getInfo =      ()
11.    {
12.      alert(this.username + ", " + this.password);

```

```

13.     }
14.
15.     Person.flag =     ;
16. }
17.}
18.
19.var p =     Person();
20.var p2 =     Person();
21.
22.p.getInfo();
23.p2.getInfo();

```

JavaScript 中的继承。

1) 对象冒充

[html] [view plaincopyprint?](#)

```

1. //继承第一种方式：对象冒充
2.
3. function Parent(username)
4. {
5.     this.username =     ;
6.
7.     this.sayHello =     ()
8.     {
9.         alert(this.username);
10.    }
11.}
12.
13.function Child(username, password)
14.{
15.    //下面三行代码是最关键的代码
16.    this.method =     ;
17.    this.method(username);

```

```

18. delete this.method;
19.
20. this.password =      ;
21.
22. this.sayWorld =      ()
23. {
24.     alert(this.password);
25. }
26.}
27.
28.var parent =      Parent("zhangsan");
29.var child =      Child("lisi", "1234");
30.
31.parent.sayHello();
32.
33.child.sayHello();
34.child.sayWorld();

```

2) call 方法方式。

call 方法是 Function 对象中的方法，因此我们定义的每个函数都拥有该方法。

可以通过函数名来调用 call 方法，call 方法的第一个参数会被传递给函数中的 this，从第 2 个参数开始，逐一赋值给函数中的参数。

[\[html\] view plaincopyprint?](#)

```

1. <P>//使用 call 方式实现对象的继承</P><P>function Parent(username)
2. {
3.     this.username =      ;</P><P> this.sayHello =      ()
4. {
5.     alert(this.username);
6. }
7. }</P><P>function Child(username, password)
8. {

```

```

9. Parent.call(this, username);</P><P> this.password =      ;</P><P> this.say
    World =      ()
10. {
11. alert(this.password);
12. }
13.</P><P>var parent =      Parent("zhangsan");</P><P>var child =      Child("lisi",
    "123");</P><P>parent.sayHello();</P><P>child.sayHello();
14.child.sayWorld();</P><P>
15. </P><SPAN style=      ><SPAN style=      > </SP
    AN></SPAN>

```

3) apply 方法方式

[html] [view plain](#)[copy](#)[print?](#)

```

1. //使用 apply 方法实现对象继承
2.
3. function Parent(username)
4. {
5.     this.username =      ;
6.
7.     this.sayHello =      ()
8.     {
9.         alert(this.username);
10.    }
11.}
12.
13.function Child(username, password)
14.{
15.    Parent.apply(this, new Array(username));
16.
17.    this.password =      ;
18.
19.    this.sayWorld =      ()
20.    {
21.        alert(this.password);

```

```

22.  }
23.}
24.
25.
26.var parent =    Parent("zhangsan");
27.var child =    Child("lisi", "123");
28.
29.parent.sayHello();
30.
31.child.sayHello();
32.child.sayWorld();

```

4) 原型链方式（无法给构造函数传参数）

[html] [view plaincopyprint?](#)

```

1. //使用原型链（prototype chain）方式实现对象继承
2.
3. function Parent()
4. {
5.
6. }
7.
8. Parent.prototype.hello =    ;
9. Parent.prototype.sayHello =    ()
10.{
11.  alert(this.hello);
12.}
13.
14.function Child()
15.{
16.
17.}
18.

```



```

19. Child.prototype = Parent();
20.
21. Child.prototype.world = ;
22. Child.prototype.sayWorld = ()
23. {
24.   alert(this.world);
25. }
26.
27. var child = Child();
28.
29. child.sayHello();
30. child.sayWorld();

```

5) 混合方式（推荐）

[html] [view plain](#)[copy](#)[print?](#)

```

1. //使用混合方式实现对象继承(推荐)
2.
3. function Parent(hello)
4. {
5.   this.hello = ;
6. }
7.
8. Parent.prototype.sayHello = ()
9. {
10.   alert(this.hello);
11. }
12.
13. function Child(hello, world)
14. {
15.   Parent.call(this, hello);
16.
17.   this.world = ;
18. }

```

```
19.  
20. Child.prototype = Parent();  
21.  
22. Child.prototype.sayWorld = ()  
23. {  
24.   alert(this.world);  
25. }  
26.  
27. var child = Child("hello", "world");  
28.  
29. child.sayHello();  
30. child.sayWorld();
```

（三十）javascript 弹出框、事件、对象化编程

一：弹出框

JavaScript 中有三种弹出框:警告(alert)、确认(confirm)以及提问(prompt)。

1.警告(alert)

在访问网站的时候，你遇到“咚”的一声，一个小窗口出现在你面前，上面写着一段警示性的文字，或是其它的提示信息。如果你不点击确定，你就不能对网页做任何的操作。没错，这个“咚”的小窗口就是 **alert** 干的。下面的代码是一段使用 **alert** 的实例。

```
<script type="text/JavaScript">
alert("我是菜鸟我怕谁");
</script>
```

2.确认(confirm)

确认框用于让用户选择某一个问题是否符合实际情况。来看下面的代码：我们用 **confirm**("你是菜鸟吗？")向访客提问，变量 **r** 则保存了访客的回应，它只可能有两种取值：**true** 或 **false**。没错，它是一个布尔值。**confirm** 后面的语句则是我们对访客回答做出的不同回应。

[html] [view plaincopyprint?](#)

```
1. <SPAN style="color:blue">var r=confirm("你是菜鸟吗");
2.
3. if (r==true)
4.
5. { document.write("彼此彼此"); }
6.
7. else
8.
9. { document.write("佩服佩服"); } </script>
10.
11. </SPAN>
```

3.提问(prompt)

prompt 和 **confirm** 类似，不过它允许访客随意输入回答。我们根据分数来做出不同的评价，现在我么就可以用 **prompt** 来向访客提问，用 **score** 存储用户输入的回答，其余的事情就都由后面的 **switch** 来完成了。

[html] [view plaincopyprint?](#)

```

1. <SPAN style=                                >function judge() {
2.
3. var score;//分数
4.
5. var degree;//分数等级
6.
7. score =                                ("你的分数是多少? ")
8.
9. if (score > 100){
10.
11.degree =                                ;}
12.
13.else{
14.
15.switch (parseInt(score / 10)) {
16.
17.case 0:
18.
19.case 1:
20.
21.case 2:
22.
23.case 3:
24.
25.case 4:
26.
27.case 5:
28.
29.degree =                                ;
30.
31.break;
32.
33.case 6:
34.
35.degree =                                ;

```

```
36.  
37.break;  
38.  
39.case 7:  
40.  
41.degree =  
42.  
43.break;  
44.  
45.case 8:  
46.  
47.degree =          ;  
48.  
49.break;  
50.  
51.case 9:  
52.  
53.case 10:  
54.  
55.degree =          ;  
56.  
57.}//end of switch  
58.  
59.}//end of else  
60.  
61.alert(degree);  
62.  
63.}  
64.  
65.</SPAN>
```

二：JavaScript 事件

我们之前提到过函数的调用。函数定义之后，默认是不会执行的，这时候就需要一些事件来触发这个函数的执行。JavaScript 很多有很多事件，例如鼠标的点击、移动，网页的载入和关闭。我们一起来看几个事件的实例。

1. 点击事件

使用点击事件调用，需要给元素设置 `onclick` 属性。示例代码如下：

```
<button value="点击提交" onclick="displaymessage()">onclick 调用函数</button>
```

由于设置了 `onclick="displaymessage()"`，因此点击按钮则会调用函数。

2. 鼠标经过、移出事件

使用鼠标经过事件调用函数的代码如下：

```
<button value="点击提交" onmouseover="displaymessage()">鼠标滑过调用函数</button>
```

当鼠标经过按钮时，触发 `onmouseover` 事件，将会调用函数 `displaymessage()`。

使用鼠标移出事件调用函数的代码如下：

```
<button value="点击提交" onmouseout="displaymessage()">鼠标移出调用函数</button>
```

把鼠标移动到这个按钮里面，当再移动出去时，触发 `onmouseout` 事件，将会调用函数 `displaymessage()`。

11.3 更多事件

JavaScript 中还有很多事件，完整的列表可以看看

http://www.w3pop.com/learn/view/p/3/o/0/doc/jsref_events/。

下面的列表列举了可以插入 HTML 标签中来定义事件动作的属性，具体的用法请参考上面的网站

属性	事件发生时机
<code>onabort</code>	图片下载被打断时
<code>onblur</code>	元素失去焦点时
<code>onchange</code>	框内容改变时
<code>onclick</code>	鼠标点击一个对象时
<code>ondblclick</code>	鼠标双击一个对象时
<code>onerror</code>	当加载文档或图片时发生错误时
<code>onfocus</code>	当元素获取焦点时
<code>onkeydown</code>	按下键盘按键时
<code>onkeypress</code>	按下或按住键盘按键时
<code>onkeyup</code>	放开键盘按键时
<code>onload</code>	页面或图片加载完成时
<code>onmousedown</code>	鼠标被按下时
<code>onmousemove</code>	鼠标被移动时

onmouseout	鼠标离开元素时
onmouseover	鼠标经过元素时
onmouseup	释放鼠标按键时
onreset	重新点击鼠标按键时
onresize	当窗口或框架被重新定义尺寸时
onselect	文本被选择时
onsubmit	点击提交按钮时
onunload	用户离开页面时

三：JavaScript 对象化编程

JavaScript 是使用“对象化编程”的，或者叫“面向对象编程”的。所谓“对象化编程”，意思是把 JavaScript 能涉及的范围划分成大大小小的对象，对象下面还继续划分对象直至非常详细为止，所有的编程都以对象为出发点，基于对象。小到一个变量，大到网页文档、窗口甚至屏幕，都是对象。JavaScript 对象是可以是一段文字、一幅图片、一个表单（Form）等等。每个对象有它自己的属性、方法和事件。对象的属性是反映该对象某些特定的性质的，例如：字符串的长度、图像的长宽、文字框（Textbox）里的文字等等；对象的方法能对该对象做一些事情，例如，表单的“提交”(Submit)，窗口的“滚动”(Scrolling)等等；而对象的事件就能响应发生在对象上的事情，例如提交表单产生表单的“提交事件”，点击连接产生的“点击事件”。不是所有的对象都有以上三个性质，有些没有事件，有些只有属性。引用对象的任一“性质”用“<对象名>.<性质名>”这种方法。

JavaScript 对象有：基本对象、全局对象、文档对象。下面我们一一介绍。

14.1 基本对象

（1）String 字符串对象：前面博客中已经提到过了，并且具体用法和方法 and java 中的基本相识，在此不在写了

（2）Array 数组对象

数组对象是一个对象的集合，里边的对象可以是不同类型的。数组的每一个成员对象都有一个“下标”，用来表示它在数组中的位置，是从零开始的。

属性

length 用法：<数组对象>.length；返回：数组的长度，即数组里有多少个元素。它等于数组里最后一个元素的下标加一。

方法

join() 用法：<数组对象>.join(<分隔符>)；返回一个字符串，该字符串把数组中的各个元素串起来，用<分隔符>置于元素与元素之间。这个方法不影响数组原本的内容。

reverse() 用法：<数组对象>.reverse()；使数组中的元素顺序反过来。如果对数组 [1, 2, 3]使用这个方法，它将使数组变成：[3, 2, 1]。

slice() 用法: `<数组对象>.slice(<始>[, <终>])`; 返回一个数组, 该数组是原数组的子集, 始于<始>, 终于<终>。如果不给出<终>, 则子集一直取到原数组的结尾。

sort() 用法: `<数组对象>.sort([<方法函数>])`; 使数组中的元素按照一定的顺序排列。如果不指定<方法函数>, 则按字母顺序排列。在这种情况下, 80 是比 9 排得前的。如果指定<方法函数>, 则按<方法函数>所指定的排序方法排序。<方法函数>比较难讲述, 这里只将一些有用的<方法函数>介绍给大家。

按升序排列数字:

```
function sortMethod(a, b) {  
  return a - b;  
}
```

```
myArray.sort(sortMethod);
```

按降序排列数字: 把上面的“a - b”该成“b - a”。

(3) Math “数学”对象

Math 对象, 提供对数据的数学计算。下面所提到的属性和方法, 不再详细说明“用法”, 大家在使用的时候记住用“**Math.<名>**”这种格式。

属性

E 返回常数 **e** (2.718281828...)。

LN2 返回 2 的自然对数 (ln 2)。

LN10 返回 10 的自然对数 (ln 10)。

LOG2E 返回以 2 为底的 **e** 的对数 (log₂e)。

LOG10E 返回以 10 为底的 **e** 的对数 (log₁₀e)。

PI 返回 π (3.1415926535...)。

SQRT1_2 返回 1/2 的平方根。 **SQRT2** 返回 2 的平方根。

方法

abs(x) 返回 **x** 的绝对值。

acos(x) 返回 **x** 的反余弦值 (余弦值等于 **x** 的角度), 用弧度表示。 **asin(x)** 返回 **x** 的反正弦值。

atan(x) 返回 **x** 的反正切值。

atan2(x, y) 返回复平面内点(x, y)对应的复数的幅角, 用弧度表示, 其值在 $-\pi$ 到 π 之间。

ceil(x) 返回大于等于 **x** 的最小整数。

cos(x) 返回 **x** 的余弦。

exp(x) 返回 **e** 的 **x** 次幂 (**ex**)。

floor(x) 返回小于等于 **x** 的最大整数。

log(x) 返回 **x** 的自然对数 (ln **x**)。

max(a, b) 返回 **a, b** 中较大的数。

min(a, b) 返回 a, b 中较小的数。

pow(n, m) 返回 n 的 m 次幂 (nm)。

random() 返回大于 0 小于 1 的一个随机数。

round(x) 返回 x 四舍五入后的值。

sin(x) 返回 x 的正弦。

sqrt(x) 返回 x 的平方根。

tan(x) 返回 x 的正切。

(4) Date 对象

Date 日期对象。这个对象可以储存任意一个日期，从 0001 年到 9999 年，并且可以精确到毫秒数（1/1000 秒）。

定义一个日期对象：

```
var today = new Date();
```

这个方法使 d 成为日期对象，并且已有初始值：当前时间。如果要自定初始值，可以用下列方法：

```
var d = new Date(99, 10, 1); //99 年 10 月 1 日  
var d = new Date('Oct 1, 1999'); //99 年 10 月 1 日
```

最好的方法就是用下面介绍的“方法”来严格的定义时间。

方法

以下有很多 **getXXX()**、**setXXX()** 这样的方法，**getXXX()** 是获得某个数值，而 **setXXX()** 是设定某个数值。

如无说明，方法的使用格式为：“<对象>.<方法>”，下同。

get/setFullYear() 返回/设置年份，用四位数表示。如果使用“**x.setFullYear(99)**”，则年份被设定为 0099 年。

get/setYear() 返回/设置年份，用两位数表示。设定的时候浏览器自动加上“19”开头，故使用“**x.setYear(00)**”把年份设定为 1900 年。

get/setMonth() 返回/设置月份，0 表示 1 月。

get/setDate() 返回/设置日期。

get/setDay() 返回/设置星期，0 表示星期天。

get/setHours() 返回/设置小时数，24 小时制。

get/setMinutes() 返回/设置分钟数。

get/setSeconds() 返回/设置秒钟数。

get/setMilliseconds() 返回/设置毫秒数。

get/setTime() 返回/设置时间，该时间就是日期对象的内部处理方法：

从 1970 年 1 月 1 日零时正开始计算到日期对象所指的日期的毫秒数。如果要使某日

期对象所指的时间推迟 1 小时，就用：“`x.setTime(x.getTime() + 60 * 60 * 1000);`”（一小时 60 分，一分 60 秒，一秒 1000 毫秒）。

`getTimezoneOffset()` 返回日期对象采用的时区与格林威治时间所差的分钟数。在格林威治东方的市区，该值为负，例如：中国时区（GMT+0800）返回

“-480”。`toString()` 返回一个字符串，描述日期对象所指的日期。这个字符串的格式类似于：“FriJul 21 15:43:46 UTC+0800 2000”。

`toLocaleString()` 返回一个字符串，描述日期对象所指的日期，用本地时间表示格式。如：“2000-07-21 15:43:46”。

`toGMTString()` 返回一个字符串，描述日期对象所指的日期，用 GMT 格式。

`toUTCString()` 返回一个字符串，描述日期对象所指的日期，用 UTC 格式。

`parse()` 用法：`Date.parse(<日期对象>);` 返回该日期对象的内部表达方式。

下面例子显示当前日期：

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                ><html>
2.
3. <body>
4.
5. <script language=          >
6.
7. today =    Date();
8.
9. var day; var date;
10.
11.if(today.getDay()==0) day =
12.
13.if(today.getDay()==1) day =
14.
15.if(today.getDay()==2) day =
16.
17.if(today.getDay()==3) day =
18.
19.if(today.getDay()==4) day =
20.
21.if(today.getDay()==5) day =
22.
```

```

23.if(today.getDay()==6) day =
24.
25.date =          + (today.getYear()) + "年" + (today.getMonth() + 1 ) + "
    月 " + today.getDate() + " 日 " + day +"";
26.
27.document.write(date);
28.
29.</script>
30.
31.</body>
32.
33.</html>
34.
35.</SPAN>

```

2.文档对象

文档对象是指在网页文档里划分出来的对象。在 JavaScript 能够涉及的范围内有如下几个“大”对象：window, document, location, navigator, screen, history 等。下面是一个文档对象树。

要引用某个对象，就要把父级的对象都列出来。例如，要引用某表单“MyForm”的某文字框“UserName”，就要用“document. MyForm. UserName”。

引用 Form 下的表单元素对象不使用名称，比如 Button，而是通过对象的 ID 或 Name 进行引用，或使用它所属的对象数组。比如：

```
<input id="UserName" type="text" />
```

```
var name = document.getElementById("UserName");//通过 id 获取值
```

（1）navigator

navigator 浏览器对象 反映了当前使用的浏览器的资料。

属性

appName 返回浏览器的“码名”，流行的 IE 和 NN 都返回 'Mozilla'。

appName 返回浏览器名。IE 返回 'Microsoft Internet Explorer'，NN 返回 'Netscape'。

appVersion 返回浏览器版本，包括了大版本号、小版本号、语言、操作平台等信息。

platform 返回浏览器的操作平台，对于 Windows 9x 上的浏览器，返回 'Win32'（大小写可能有差异）。

userAgent 返回以上全部信息。例如，IE5.01 返

回 'Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)'。

javaEnabled() 返回一个布尔值，代表当前浏览器允许不允许 Java。

(2) screen

screen 屏幕对象 反映了当前用户的屏幕设置。

属性

width 返回屏幕的宽度（像素数）。 **height** 返回屏幕的高度。 **availWidth** 返回屏幕的可用宽度（除去了一些不自动隐藏的类似任务栏的东西所占用的宽度）。

availHeight 返回屏幕的可用高度。

colorDepth 返回当前颜色设置所用的位数 - 1：黑白；8：256 色；16：增强色；24/32：真彩色

(3) window

window 窗口对象是最大的对象，它描述的是一个浏览器窗口。一般要引用它的属性和方法时，不需要用“**window.xxx**”这种形式，而直接使用“**xxx**”。一个框架页面也是一个窗口。

属性

name 窗口的名称，由打开它的连接（）或框架页（<frame name="...">）或某一个窗口调用的 **open()** 方法决定。一般我们不会用这个属性。

status 指窗口下方的“状态栏”所显示的内容。通过对 **status** 赋值，可以改变状态栏的显示。

opener 用法： **window.opener**；返回打开本窗口的窗口对象。

注意：返回的是一个窗口对象。如果窗口不是由其他窗口打开的，在 Netscape 中这个属性返回 **null**；在 IE 中返回“未定义”（**undefined**）。

undefined 在一定程度上等于 **null**。

注意：**undefined** 不是 JavaScript 常数，如果你企图使用“**undefined**”，那就真的返回“未定义”了。

self 指窗口本身，它返回的对象跟 **window** 对象是一模一样的。最常用的是

“**self.close()**”，放在<a>标记中：“关闭窗口”。

parent 返回窗口所属的框架页对象。

top 返回占据整个浏览器窗口的最顶端的框架页对象。

history 历史对象，见下。

location 地址对象，见下。

document 文档对象，见下。

方法

open() 打开一个窗口。

用法: `open(<URL 字符串>, <窗口名称字符串>, <参数字符串>);` <URL 字符串>: 描述所打开的窗口打开哪一个网页。如果留空(""), 则不打开任何网页。 <窗口名称字符串>: 描述被打开的窗口的名称 (`window.name`), 可以使用'_top'、'_blank'等内建名称。这里的名称跟""里的"target"属性是一样的。 <参数字符串>: 描述被打开的窗口的样式。如果只需要打开一个普通窗口, 该字符串留空(""), 如果要指定样式, 就在字符串里写上一到多个参数, 参数之间用逗号隔开。

例: 打开一个 400 x 100 的干净的窗

口: `open("','_blank','width=400,height=100,menubar=no,toolbar=no, location=no,directories=no,status=no,scrollbars=yes,resizable=yes')`

参数

<code>top=#</code>	窗口顶部离开屏幕顶部的像素数
<code>left=#</code>	窗口左端离开屏幕左端的像素数
<code>width=#</code>	窗口的宽度
<code>height=#</code>	窗口的高度
<code>menubar=...</code>	窗口有没有菜单, 取值 yes 或 no
<code>toolbar=...</code>	窗口有没有工具条, 取值 yes 或 no
<code>location=...</code>	窗口有没有地址栏, 取值 yes 或 no
<code>directories=...</code>	窗口有没有连接区, 取值 yes 或 no
<code>scrollbars=...</code>	窗口有没有滚动条, 取值 yes 或 no
<code>status=...</code>	窗口有没有状态栏, 取值 yes 或 no
<code>resizable=...</code>	窗口给不给调整大小, 取值 yes 或 no
<code>fullscreen=</code>	窗口是否全屏, 取值 yes 或 no

`open()` 方法有返回值, 返回的就是它打开的窗口对象。所以,

```
var newWindow = open("','_blank');
```

这样把一个新窗口赋值到“newWindow”变量中, 以后通过“newWindow”变量就可以控制窗口了。

`close()`关闭一个已打开的窗口。

用法: `window.close()` 或 `self.close()`: 关闭本窗口; `<窗口对象>.close()`: 关闭指定的窗口。如果该窗口有状态栏, 调用该方法后浏览器会警告: “网页正在试图关闭窗口, 是否关闭?”然后等待用户选择是否; 如果没有状态栏, 调用该方法将直接关闭窗口。

`blur()` 使焦点从窗口移走, 窗口变为“非活动窗口”。`focus()` 是窗口获得焦点, 变为“活动窗口”。

`scrollTo()` 用法: `[<窗口对象>].scrollTo(x, y);` 使窗口滚动, 使文档从左上角数起的(x, y)点滚动到窗口的左上角。

`scrollBy()` 用法: `[<窗口对象>].scrollBy(deltaX, deltaY);` 使窗口向右滚动 `deltaX` 像素, 向下滚动 `deltaY` 像素。如果取负值, 则向相反的方向滚动。

resizeTo() 用法: [**<窗口对象>**].**resizeTo**(width, height); 使窗口调整大小到宽 width 像素, 高 height 像素。

resizeBy() 用法: [**<窗口对象>**].**resizeBy**(deltaWidth, deltaHeight); 使窗口调整大小, 宽增大 deltaWidth 像素, 高增大 deltaHeight 像素。如果取负值, 则减少。

alert() 用法: **alert**(<字符串>); 弹出一个只包含“确定”按钮的对话框, 显示<字符串>的内容, 整个文档的读取、Script 的运行都会暂停, 直到用户按下“确定”。

confirm() 用法: **confirm**(<字符串>); 弹出一个包含“确定”和“取消”按钮的对话框, 显示<字符串>的内容, 要求用户做出选择, 整个文档的读取、Script 的运行都会暂停。如果用户按下“确定”, 则返回 true 值, 如果按下“取消”, 则返回 false 值。

prompt() 用法: **prompt**(<字符串>[, <初始值>]); 弹出一个包含“确认”“取消”和一个文本框的对话框, 显示<字符串>的内容, 要求用户在文本框输入一些数据, 整个文档的读取、Script 的运行都会暂停。如果用户按下“确认”, 则返回文本框里已有的内容, 如果用户按下“取消”, 则返回 null 值。如果指定<初始值>, 则文本框里会有默认值。

setTimeout()和**setInterval()**的使用

这两个方法都可以用来实现在一个固定时间段之后去执行 JavaScript。不过两者各有各的应用场景。实际上, **setTimeout** 和 **setInterval** 的语法相同。它们都有两个参数, 一个是将要执行的代码字符串, 还有一个是以毫秒为单位的时间间隔, 当过了那个时间段之后就将执行那段代码。不过这两个函数还是有区别的, **setInterval** 在执行完一次代码之后, 经过了那个固定的时间间隔, 它还会自动重复执行代码, 而 **setTimeout** 只执行一次那段代码。虽然表面上看来 **setTimeout** 只能应用在 on-off 方式的动作上, 不过可以通过创建一个函数循环重复调用 **setTimeout**, 以实现重复的操作:

```
showTime();
function showTime()
{
var today = new Date();
alert("The time is: " + today.toString());
setTimeout("showTime()", 5000);
}
```

一旦调用了这个函数, 那么就会每隔 5 秒钟就显示一次时间。如果使用 **setInterval**, 则相应的代码如下所示:

```
setInterval("showTime()", 5000);
function showTime()
{
var today = new Date();
alert("The time is: " + today.toString());
}
```

```
}
```

这两种方法可能看起来非常像，而且显示的结果也会很相似，不过两者的最大区别就是，**setTimeout** 方法不会每隔 5 秒钟就执行一次 **showTime** 函数，它是在每次调用 **setTimeout** 后过 5 秒钟再去执行 **showTime** 函数。这意味着如果 **showTime** 函数的主体部分需要 2 秒钟执行完，那么整个函数则要每 7 秒钟才执行一次。而 **setInterval** 却没有被自己所调用的函数所束缚，它只是简单地每隔一定时间就重复执行一次那个函数。如果要求在每隔一个固定的时间间隔后就精确地执行某动作，那么最好使用 **setInterval**，而如果不希望由于连续调用产生互相干扰的问题，尤其是每次函数的调用需要繁重的计算以及很长的处理时间，那么最好使用 **setTimeout**。用 **setInterval** 命令来创建的对象，可以用 **clearInterval()** 命令来终止。比如：

```
var MyMar=setInterval(showTime(),speed);
clearInterval(MyMar);
```

(4) history

history 历史对象指浏览器的浏览历史。

属性

length 历史的项数。**JavaScript** 所能管到的历史被限制在用浏览器的“前进”“后退”键可以去到的范围。本属性返回的是“前进”和“后退”两个按键之下包含的地址数的和。

方法

back() 后退，跟按下“后退”键是等效的。

forward() 前进，跟按下“前进”键是等效的。

go() 用法：**history.go(x)**；在历史的范围内去到指定的一个地址。如果 $x < 0$ ，则后退 x 个地址，如果 $x > 0$ ，则前进 x 个地址，如果 $x == 0$ ，则刷新现在打开的网页。

history.go(0) 跟 **location.reload()** 是等效的。

(5) location

location 地址对象描述的是某一个窗口对象所打开的地址。要表示当前窗口的地址，只需要使用“**location**”就行了；若要表示某一个窗口的地址，就使用“<窗口对象>.**location**”。

注意 属于不同协议或不同主机的两个地址之间不能互相引用对方的 **location** 对象，这是出于安全性的需要。例如，当前窗口打开的是“**www.a.com**”下面的某一页，另外一个窗口（对象名为：**bWindow**）打开的是“**www.b.com**”的网页。如果在当前窗口使用“**bWindow.location**”，就会出错：“没有权限”。这个错误是不能用错误处理程序（**Event Handler**，参阅 **onerror** 事件）来接收处理的。

（**Event Handler**，参阅 **onerror** 事件）来接收处理的。

属性

protocol 返回地址的协议，取值为 '**http:**','**https:**','**file:**' 等等。**hostname** 返回地址的主机名，例如，一个“**http://www.microsoft.com/china/**”的地址，

`location.hostname == 'www.microsoft.com'`。 `port` 返回地址的端口号，一般 `http` 的端口号是 '80'。 `host` 返回主机名和端口号，如： `'www.a.com:8080'`。 `pathname` 返回路径名，如 `"http://www.a.com/b/c.html"`， `location.pathname == 'b/c.html'`。 `hash` 返回“#”以及以后的内容，如 `"http://www.a.com/b/c.html#chapter4"`，
`location.hash == '#chapter4'`；如果地址里没有“#”，则返回空字符串。 `search` 返回“?”以及以后的内容，如 `"http://www.a.com/b/c.asp?selection=3&jump=4"`，
`location.search == '?selection=3&jump=4'`；如果地址里没有“?”，则返回空字符串。 `href` 返回以上全部内容，也就是说，返回整个地址。在浏览器的地址栏上怎么显示它就怎么返回。如果想一个窗口对象打开某地址，可以使用 `"location.href = '...'"`，也可以直接用 `"location = '...'"` 来达到此目的。

方法

`reload()` 相当于按浏览器上的“刷新”(IE)或“Reload”(Netscape)键。 `replace()` 打开一个 URL，并取代历史对象中当前位置的地址。用这个方法打开一个 URL 后，按下浏览器的“后退”键将不能返回到刚才的页面。

(6) document

`document` 文档对象 描述当前窗口或指定窗口对象的文档。它包含了文档从 `<head>` 到 `</body>` 的内容。用法：`document`（当前窗口）或 `<窗口对象>.document`（指定窗口）

属性

`lastModified` 当前文档的最后修改日期，是一个 `Date` 对象。 `referrer` 如果当前文档是通过点击连接打开的，则 `referrer` 返回原来的 URL。

`title` 指 `<head>` 标记里用 `<title>...</title>` 定义的文字。在 Netscape 里本属性不接受赋值。

`fgColor` 指 `<body>` 标记的 `text` 属性所表示的文本颜色。

`bgColor` 指 `<body>` 标记的 `bgcolor` 属性所表示的背景颜色。 `linkColor` 指 `<body>` 标记的 `link` 属性所表示的连接颜色。 `alinkColor` 指 `<body>` 标记的 `alink` 属性所表示的活动连接颜色。 `vlinkColor` 指 `<body>` 标记的 `vlink` 属性所表示的已访问连接颜色。

方法

`open()` 打开文档以便 JavaScript 能向文档的当前位置（指插入 JavaScript 的位置）写入数据。通常不需要用这个方法，在需要的时候 JavaScript 自动调用。

`write()`; `writeln()` 向文档写入数据，所写入的会当成标准文档 HTML 来处理。

`writeln()` 与 `write()` 的不同点在于，`writeln()` 在写入数据以后会加一个换行。这个换行只是在 HTML 中换行，具体情况能不能够是显示出来的文字换行，要看插入 JavaScript 的位置而定。如在 `<pre>` 标记中插入，这个换行也会体现在文档中。

`clear()` 清空当前文档。

`close()` 关闭文档，停止写入数据。如果用了 `write[ln]()` 或 `clear()` 方法，就一定要用 `close()` 方法来保证所做的更改能够显示出来。如果文档还没有完全读取，也就是说，JavaScript 是插在文档中的，那就不必使用该方法。

现在我们已经拥有足够的知识来做以下这个很多网站都有的弹出式更新通知了。

```
<script language="JavaScript"> <!-- var whatsNew = open('', '_blank', 'top=50, left=50, width=200, height=300, ' + 'menubar=no, toolbar=no, directories=no, location=no, ' + 'status=no, resizable=no, scrollbars=yes');
whatsNew.document.write('<center><b>更新通知</b></center>');
whatsNew.document.write('<p>最后更新日期: 00.08.01');
whatsNew.document.write('<p>00.08.01: 增加了“我的最爱”栏目。');
whatsNew.document.write('<p align="right">' + '<a href="javascript:self.close()">关闭窗口</a>');
whatsNew.document.close(); --> </script>
```

当然也可以先写好一个 HTML 文件，在 `open()` 方法中直接 load 这个文件。

(7) `anchors[]`; `links[]`; `Link`

`anchors[]`; `links[]`; `Link` 连接对象。

用法: `document.anchors[[x]]`; `document.links[[x]]`; `<anchorId>`; `<linkId>`

`document.anchors` 是一个数组，包含了文档中所有锚标记（包含 `name` 属性的 `<a>` 标记），按照在文档中的次序，从 0 开始给每个锚标记定义了一个下标。

`document.links` 也是一个数组，包含了文档中所有连接标记（包含 `href` 属性的 `<a>` 标记和 `<map>` 标记段里的 `<area>` 标记），按照在文档中的次序，从 0 开始给每个连接标记定义了一个下标。如果一个 `<a>` 标记既有 `name` 属性，又有 `href` 属性，则它既是一个 `Anchor` 对象，又是一个 `Link` 对象。在 IE 中，如果在 `<a>` 标记中添加 `id="..."` 属性，则这个 `<a>` 对象被赋予一个标识（ID），调用这个对象的时候只需要使用 `<id>` 就行了。很多文档部件都可以用这个方法赋予 ID，但要注意不能有两个 ID 相同。

`anchors` 和 `links` 作为数组，有数组的属性和方法。单个 `Anchor` 对象没有属性；单个 `Link` 对象的属性见下。

属性

`protocol`; `hostname`; `port`; `host`; `pathname`; `hash`; `search`; `href` 与 `location` 对象相同。`target` 返回/指定连接的目标窗口（字符串），与 `<a>` 标记里的 `target` 属性是一样的。

（三十一）大话设计模式（一）设计模式遵循的七大原则

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

最近几年来，人们踊跃的提倡和使用设计模式，其根本原因就是为了解现代代码的复用性，增加代码的可维护性。设计模式的实现遵循了一些原则，从而达到代码的复用性及增加可维护性的目的，设计模式对理解面向对象对象的三大特征有很好的启发，不看设计模式，很难深层地体会到面向对象开发带来的好处。在刚开始学习中，很难做到将这些模式融汇贯通，所以这个需要我们在编码前多思考，等想充分了，在开始实践编码。下面是设计模式应当遵循的七大原则

1.开闭原则（Open Close Principle）

定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

开放-封闭原则的意思就是说，你设计的时候，时刻要考虑，尽量让这个类是足够好，写好了就不要去修改了，如果新需求来，我们增加一些类就完事了，原来的代码能不动则不动。这个原则有两个特性，一个是说“对于扩展是开放的”，另一个是说“对于更改是封闭的”。面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。这就是“开放-封闭原则”的精神所在

比如，刚开始需求只是写加法程序，很快在 `client` 类中完成后，此时变化没有发生，需求让再添加一个减法功能，此时会发现增加功能需要修改原来这个类，这就违背了开放-封闭原则，于是你就应该考虑重构程序，增加一

个抽象的运算类，通过一些面向对象的手段，如继承、动态等来隔离具体加法、减法与 **client** 耦合，需求依然可以满足，还能应对变化。此时需求要添加乘除法功能，就不需要再去更改 **client** 及加减法类，而是增加乘法和除法子类即可。

绝对的修改关闭是不可能的，无论模块是多么的‘封闭’，都会存在一些无法对之封闭的变化，既然不可能完全封闭，设计人员必须对于他设计的模块应该对哪种变化封闭做出选择。他必须先猜测出最有可能发生的变化种类，然后构造抽象来隔离那些变化。在我们最初编写代码时，假设变化不会发生，当变化发生时，我们就创建抽象来隔离以后发生同类的变化。

我们希望的是在开发工作展开不久就知道可能发生的变化，查明可能发生的变化所等待的时候越长，要创建正确的抽象就越困难。开放-封闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术所声称的巨大好处，也就是可维护、可扩展、可复用、灵活性好。开发人员应该仅对程序中呈现出频繁变化的那些部分做出抽象，然而对于应用程序中的每个部分都刻意地进行抽象同样不是一个好主意，拒绝不成熟的抽象和抽象本身一样重要。开放-封闭原则，可以保证以前代码的正确性，因为没有修改以前代码，所以可以保证开发人员专注于将设计放在新扩展的代码上。

简单的用一句经典的话来说：过去的事已成历史，是不可修改的，因为时光不可倒流，但现在或明天计划做什么，是可以自己决定(即扩展)的。

2.里氏代换原则（**Liskov Substitution Principle**）

定义 1: 如果对每一个类型为 **T1** 的对象 **o1**，都有类型为 **T2** 的对象 **o2**，使得以 **T1** 定义的所有程序 **P** 在所有的对象 **o1** 都代换成 **o2** 时，程序 **P** 的行为没有发生变化，那么类型 **T2** 是类型 **T1** 的子类型。

定义 2: 子类型必须能够替换掉它们的父类型。

描述：一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别，也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化

例子：在生物学分类上，企鹅是一种鸟，但在编程世界里，企鹅却不能继承鸟。在面向对象设计时，子类拥有父类所有非 **private** 的行为和属性，鸟会飞，但企鹅不会飞，所以企鹅不能继承鸟类。

只有当子类可以替换掉父类，软件单位的功能不受影响时，父类才能真正被复用，而子类也能够在父类的基础上增加新的行为，正是有里氏代换原则，使得继承复用成为了可能。正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展，不然还谈什么扩展开放，修改关闭呢

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下 4 层含义：

- 1.子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
- 2.子类中可以增加自己特有的方法。
- 3.当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。

4.当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

看上去很不可思议，因为我们会发现在自己编程中常常会违反里氏替换原则，程序照样跑的好好的。所以大家都会产生这样的疑问，假如我非要遵循里氏替换原则会有什么后果？

后果就是：你写的代码出问题的几率将会大大增加。

3.依赖倒转原则（**Dependence Inversion Principle**）

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。即针对接口编程，不要针对实现编程

依赖倒转其实就是谁也不要依靠谁，除了约定的接口，大家都可以灵活自如。依赖倒转可以说是面向对象设计的标志，用哪种语言来编写程序不重要，如果编写时考虑的都是如何针对抽象编程而不是针对细节编程，即程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程化的设计了。如果设计的各个部件或类相互依赖，这样就是耦合度高，难以维护和扩展，这也就体现不出面向对象的好处了。

依赖倒转原则，好比一个团队，有需求组，开发组，测试组，开发组和测试组都是面对同样的需求后，做自己相应的工作，而不应该是测试组按照开发组理解的需求去做测试用例，也就是说开发组和测试组都是直接面向需求组工作，大家的目的是是一样的，保证产品按时上线，需求是不依赖于开发和测试的。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在 **java** 中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

依赖倒置原则的中心思想是面向接口编程，传递依赖关系有三种方式，以上的说的是接口传递，另外还有两种传递方式：构造方法传递和 **setter** 方法传递，相信用过 **Spring** 框架的，对依赖的传递方式一定不会陌生。

在实际编程中，我们一般要做到如下 **3** 点：

低层模块尽量都要有抽象类或接口，或者两者都有。

变量的声明类型尽量是抽象类或接口。

使用继承时遵循里氏替换原则。

总之，依赖倒置原则就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

4.接口隔离原则（**Interface Segregation Principle**）

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉得接口隔离原则跟单一职责原则很相似，其实不然。其一，单一职责原则注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

采用接口隔离原则对接口进行约束时，要注意以下几点：

1. 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不争的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
2. 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
3. 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

4.组合/聚合复用原则

就是要尽量地使用合成和聚合，而不是继承关系达到复用的目的

该原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分：新的对象通过向这些对象的委派达到复用已有功能的目的。

其实这里最终要的地方就是区分 “has-a” 和 “is-a” 的区别。相对于合成和聚合，

继承的缺点在于：父类的方法全部暴露给子类。父类如果发生变化，子类也

得发生变化。聚合的复用的时候就对另外的类依赖的比较少。。

合成/聚合复用

① 优点：

新对象存取成分对象的唯一方法是通过成分对象的接口；

这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的；

这种复用支持包装；

这种复用所需的依赖较少；

每一个新的类可以将焦点集中在一个任务上；

这种复用可以在运行时动态进行，新对象可以使用合成/聚合关系将新的责任委派到合适的对象。

② 缺点：

通过这种方式复用建造的系统会有较多的对象需要管理。

继承复用

① 优点：

新的实现较为容易，因为基类的大部分功能可以通过继承关系自动进入派生类；

修改或扩展继承而来的实现较为容易。

② 缺点：

继承复用破坏包装，因为继承将基类的实现细节暴露给派生类，这种复用也称为白箱复用；

如果基类的实现发生改变，那么派生类的实现也不得不发生改变；

从基类继承而来的实现是静态的，不可能在运行时发生改变，不够灵活。

6.迪米特法则 (Law Of Demeter)

迪米特法则其根本思想，是强调了类之间的松耦合，类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成影响，也就是说，信息的隐藏促进了软件的复用。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在 1987 年由美国 Northeastern University 的 Ian Holland 提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地将逻辑封装在类的内部，对外除了提供的 **public** 方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

一句话总结就是：一个对象应该对其他对象保持最少的了解。

7.单一职责原则 (Single Responsibility Principle)

定义：不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责，应该仅有一个引起它变化的原因

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责 **P** 被分化为粒度更细的职责 **P1** 和 **P2**。

遵循单一职责原则的优点有：

- 1.可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
- 2.提高类的可读性，提高系统的可维护性；
- 3.变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都需要遵循这一重要原则。

（三十二）大话设计模式（二）设计模式分类和三种工厂模式

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

设计模式分类

首先先简单说一下设计模式的分类设计模式可以分为三大类，分别是**创建型设计模式**、**行为型设计模式**以及**结构型设计模式**。

创建型的设计模式：单例模式(Singleton)、构建模式(Builder)、原型模式(Prototype)、抽象工厂模式(Abstract Factory)、工厂方法模式(Factory Method)

行为设计模式：策略模式(Strategy)、状态模式(State)、责任链模式(Chain of Responsibility)、解释器模式(Interpreter)、命令模式(Command)、观察者模式(Observer)、备忘录模式(Memento)、迭代器模式(Iterator)、模板方法模式(Template Method)、访问者模式(Visitor)、中介者模式(Mediator)

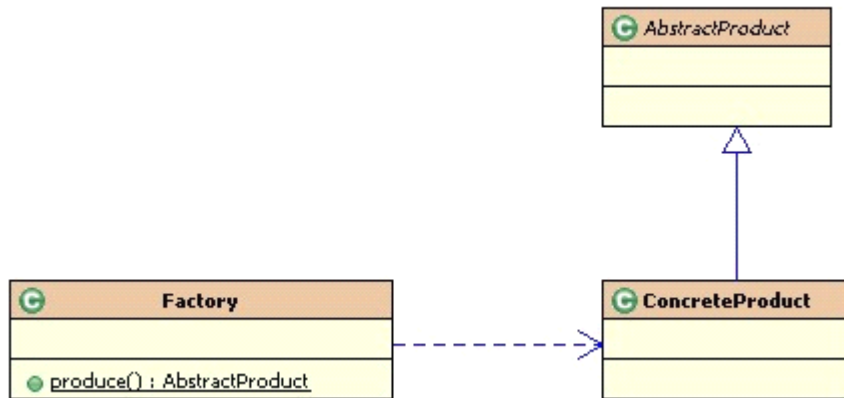
结构型设计模式：装饰者模式(Decorator)、代理模式(Proxy)、组合模式(Composite)、桥连接模式(Bridge)、适配器模式(Adapter)、蝇量模式(Flyweight)、外观模式(Facade)

简单工厂模式

从设计模式的类型上来说，简单工厂模式是属于创建型模式，又叫做静态工厂方法（StaticFactory Method）模式，但不属于 23 种 GOF 设计模式之一。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单

工厂模式是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现。

简单工厂模式的一般结构，如图所示：



简单工厂模式存在三个组成部分，参考《Java 与模式》一书，对应于三个不同的角色：

I 工厂角色

I 抽象产品角色

I 具体产品角色

其实角色这个词用的比较确切，能够让我们理解到，每个角色的不是单纯地指一个类，可能是一组类所构成了这个角色。下面对三个角色进行描述：

1. 工厂角色

工厂角色负责产品的生产工作。在简单工厂模式中，工厂类是一个具体的实现类，在系统设计中工厂类负责实际对象的创建工作。

工厂类（**Factory**）的特点是：它知道系统中都存在哪些能够创建对象的具体类（**ConcreteProduct**），也知道该如何将创建的对象，以某种能够屏蔽具体类实现细节的方式（**AbstractProduct**）提供给所需要的其他角色来使用该对象提供的数据和服务。

2.抽象产品角色

抽象产品角色是具体的产品的抽象。抽象就是将产品的共性抽取出来，可以直接暴露给客户端（需要使用具体产品的角色），对所有的客户端来说，从工厂中直接获取到的原始产品的外部形态都是相同的，没有任何的差别，包括数据和服务。这也就是说，具体客户端应该“秘密”掌握着某一个或一些具体产品的详细资料（具体产品类型、数据和服务），然后根据具体客户端（任何一个需要使用某种具体产品的数据和服务的实现类）需要什么样的附加数据和服务，进行类类型转换后，通过借助于对应的具体产品对象来完成其职责。

抽象产品角色，在实际系统中可以定义为接口或者抽象类。

3.具体产品角色

具体产品实现类一定是抽象产品类的实现或扩展。为了保证工厂类能够创建对象，工厂类需要知道具体产品的创建方式，这就涉及到具体产品类所提供的构造方法，以便，可能工厂类会向客户端提供具体创建服务所需要的数据。例如：某个产品类需要通过一个账号才能构造其实例，所以工厂类必须根据它的创建需求，为客户端提供一个带账号参数的生产方法，才能创建该具体产品类的对象。

也就是说，工厂类依赖于具体产品实现类。同样，客户端类是依赖于工厂类的。

通过上述三个角色的描述，我们应该能够了解，系统中哪些类能够胜任上述的三个角色，并通过各类之间的关系，通过工厂模式来实现系统或者某个模块。在实际的设计过程中，可能不存在完全与上述基本简单工厂模式完全适

应的，需要根据具体的需求来调整简单工厂模式的应用。只要能够实现系统的良好设计，有时候变化才能满足需要。

下面用一个简单的例子来说明一下，给大家加深一下印象(例子来自于网络)：

[html] [view plaincopyprint?](#)

```
1. 运动员.java
2. public interface 运动员 {
3.     public void 跑();
4.     public void 跳();
5. }
6. 足球运动员.java
7. public class 足球运动员 implements 运动员 {
8.     public void 跑(){
9.         //跑啊跑
10.    }
11.    public void 跳(){
12.        //跳啊跳
13.    }
14.}
15.篮球运动员.java
16.public class 篮球运动员 implements 运动员 {
17.    public void 跑(){
18.        //do nothing
19.    }
20.    public void 跳(){
21.        //do nothing
22.    }
23.}
24.体育协会.java
25.public class 体育协会 {
26.    public static 运动员 注册足球运动员(){
27.        return new 足球运动员();
28.    }
29.    public static 运动员 注册篮球运动员(){
```



```

30.         return new 篮球运动员();
31.     }
32.}
33.
34.俱乐部.java
35.public class 俱乐部 {
36.    private 运动员 守门员;
37.    private 运动员 后卫;
38.    private 运动员 前锋;
39.    public void test() {
40.        this.前锋 = 体育协会.注册足球运动员();
41.        this.后卫 = 体育协会.注册足球运动员();
42.        this.守门员 = 体育协会.注册足球运动员();
43.
44.        守门员.跑();
45.        后卫.跳();
46.    }
47.}

```

以上就是简单工厂模式的一个简单实例，读者应该想象不用接口不用工厂而把具体类暴露给客户端的那种混乱情形吧（就好像没了体育总局，各个俱乐部在市场上自己胡乱的寻找仔细需要的运动员），简单工厂就解决了这种混乱。

工厂方法模式

工厂方法模式是类的创建模式，又叫虚拟构造子（**Virtual Constructor**）模式或者多态性工厂（**Polymorphic Factory**）模式。工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际工作推迟到子类中。

工厂方法模式是[简单工厂模式](#)的衍生，解决了许多简单工厂模式的问题。首先完全实现‘开一闭 原则’，实现了可扩展。其次更复杂的层次结构，可以应用于产品结果复杂的场合。工厂方法模式的对简单工厂模式进行了抽象。有一个抽象的 **Factory** 类（可以是抽象类和接口），这个类将不在负责具体的

产品生产，而是只制定一些规范，具体的生产工作由其子类去完成。在这个模式中，工厂类和产品类往往可以依次对应。即一个抽象工厂对应一个抽象产品，一个具体工厂对应一个具体产品，这个具体的工厂就负责生产对应的产品。

工厂方法模式角色与结构

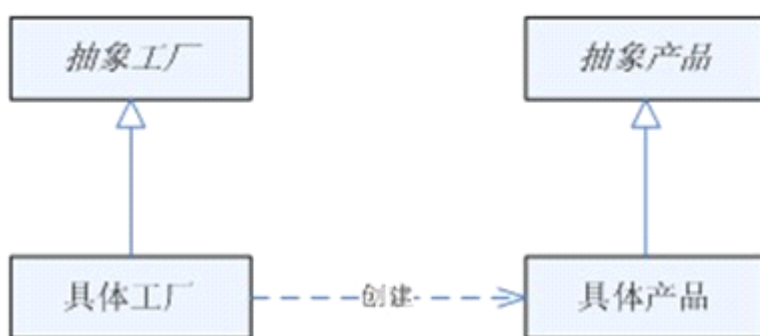
1.抽象工厂(Creator)角色：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象工厂类必须实现这个接口。

2.具体工厂(Concrete Creator)角色：这是实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建产品对象。

3.抽象产品(Product)角色：工厂方法模式所创建的对象超类型，也就是产品对象的共同父类或共同拥有的接口。

4.具体产品(Concrete Product)角色：这个角色实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。

工厂方法模式的一般结构，如图所示：



我们在不改变产品类（“足球运动员”类和“篮球运动员”类）的情况下，写一下工厂方法模式的例子：

[\[html\] view plaincopyprint?](#)

1. 运动员.java

```

2. public interface 运动员 {
3.     public void 跑();
4.     public void 跳();
5. }
6.
7. 足球运动员.java
8. public class 足球运动员 implements 运动员 {
9.
10.    public void 跑(){
11.        //跑啊跑
12.    }
13.
14.    public void 跳(){
15.        //跳啊跳
16.    }
17.}
18.
19. 篮球运动员.java
20. public class 篮球运动员 implements 运动员 {
21.
22.    public void 跑(){
23.        //do nothing
24.    }
25.
26.    public void 跳(){
27.        //do nothing
28.    }
29.}
30.
31. 体育协会.java
32. public interface 体育协会 {
33.     public 运动员 注册();
34.}
35.
36. 足球协会.java

```

```

37. public class 足球协会 implements 体育协会 {
38.     public 运动员 注册(){
39.         return new 足球运动员();
40.     }
41. }
42.
43. 篮球协会.java
44. public class 篮球协会 implements 体育协会 {
45.     public 运动员 注册(){
46.         return new 篮球运动员();
47.     }
48. }
49.
50. 俱乐部.java
51. public class 俱乐部 {
52.     private 运动员 守门员;
53.     private 运动员 后卫;
54.     private 运动员 前锋;
55.
56.     public void test() {
57.         体育协会 中国足协 = new 足球协会();
58.
59.         this.前锋 = 中国足协.注册();
60.         this.后卫 = 中国足协.注册();
61.
62.         守门员.跑();
63.         后卫.跳();
64.     }
65. }

```

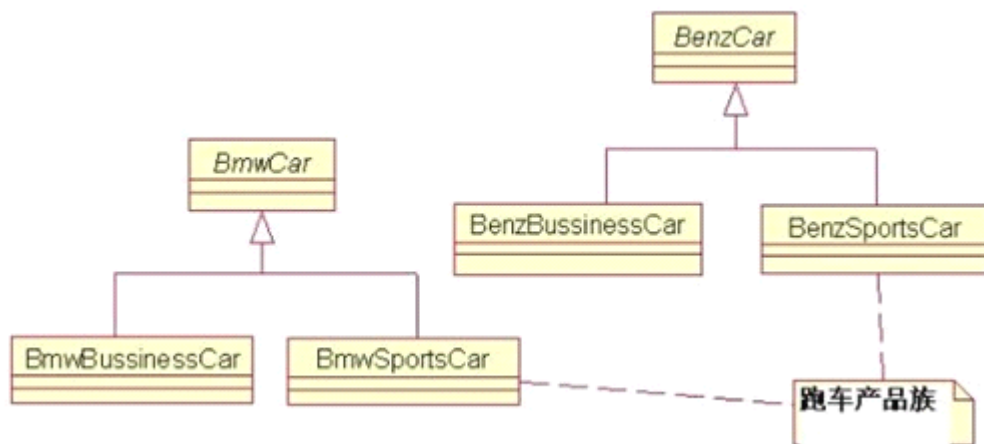
很明显可以看到，“体育协会”工厂类变成了“体育协会”接口，而实现此接口的分别是“足球协会”“篮球协会”等等具体的工厂类。

这样做有什么好处呢？很明显，这样做就完全 OCP 了。如果需要再加入（或扩展）产品类（比如加多个“乒乓球运动员”）的话就不再需要修改工厂类了，而只需相应的再添加一个实现了工厂接口（“体育协会”接口）的具体工厂类。

抽象工厂模式

抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。根据 LSP 原则，任何接受父类型的地方，都应当能够接受子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换言之，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例。

先来认识下什么是产品族：位于不同产品等级结构中，功能相关联的产品组成的家族。还是让我们用一个例子来形象地说明一下吧。



图中的 BmwCar 和 BenzCar 就是两个产品树（产品层次结构）；而如图所示的 BenzSportsCar 和 BmwSportsCar 就是一个产品族。他们都可以放到跑车家族中，因此功能有所关联。同理 BmwBussinessCar 和 BenzSportsCar 也是一个产品族。

抽象工厂模式中的有以下的四种角色：

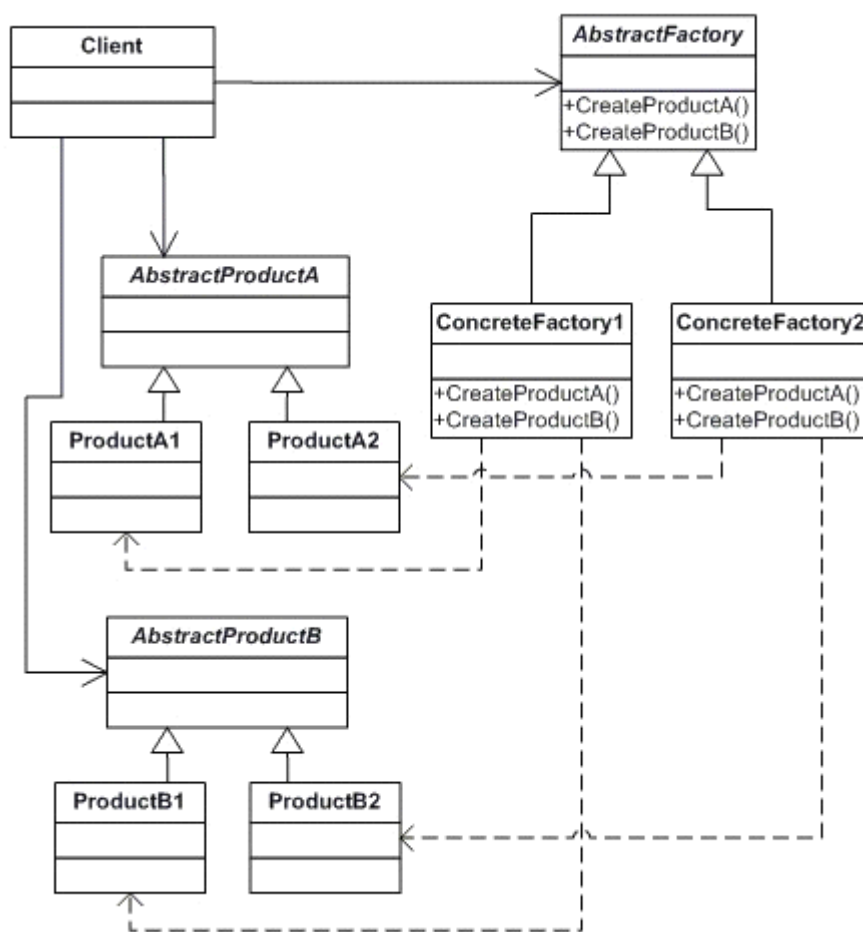
抽象工厂（Abstract Factory）角色：担任这个角色的是工厂方法模式的核心，它是与应用系统商业逻辑无关的。

具体工厂（Concrete Factory）角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。

抽象产品（Abstract Product）角色：担任这个角色的类是工厂方法模式所创建的对象之父类，或它们共同拥有的接口。

具体产品（Concrete Product）角色：这个角色用以代表具体的产品。

Abstract Factory 模式的结构：



抽象工厂模式就相当于创建实例对象的 **new**，由于经常要根据类生成实例对象，抽象工厂模式也是用来创建实例对象的，所以在需要新的实例对象时便可以考虑是否使用工厂模式。虽然这样做可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。

举例来说：生产餐具和相应食物的工厂，有两个车间，其中一个车间用以生产餐具，一个车间用以生产相应的食物。

当消费者消费时，只需要向相应的具体工厂请求具体餐具和具体食物便可以使用餐具消费食物。

使用 UML 图表示以上的描述如下：

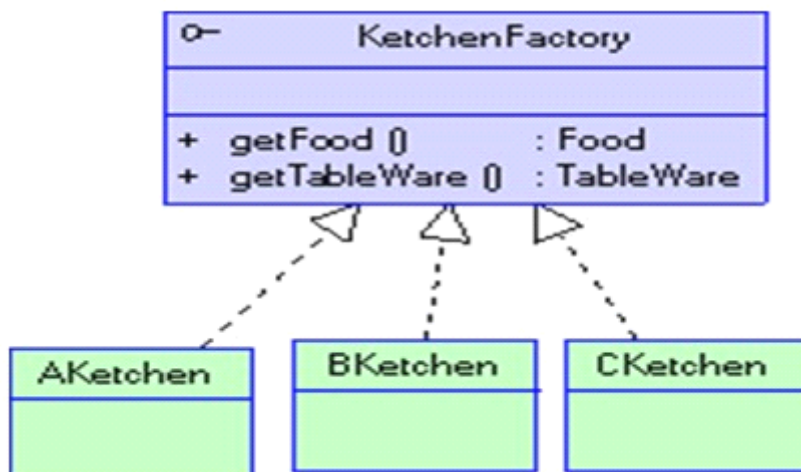


图 1 抽象工厂与具体工厂

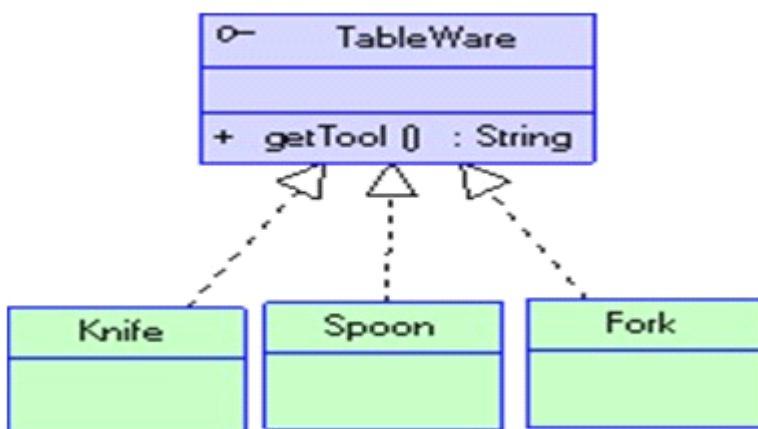


图 2 抽象餐具与具体餐具（生产车间）

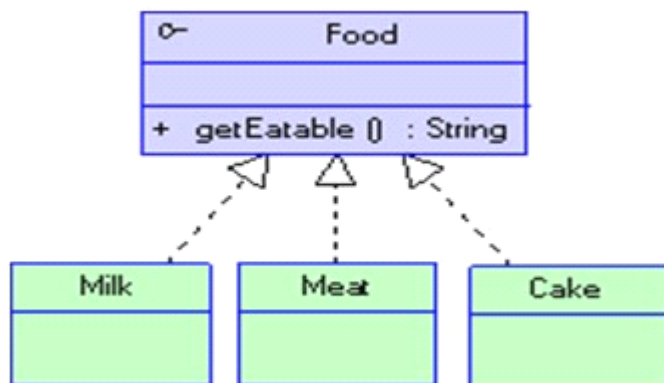


图 3 抽象食物与具体食物

注：图中厨房单词写错了：kitchen

每个具体工厂生产出来的具体产品根据不同工厂的不同各不相同，但是客户使用产品的方法是一致的。比如客户在得到餐具和食物之后，两者的搭配是正确的（使用汤匙喝牛奶，使用刀子切面包）。

在本例子中有 3 个具体工厂 **AKitchen**，**BKitchen**，**BKitchen**，分别生产牛奶和汤匙、面包和刀、肉和叉子。牛奶、面包和肉都实现了食物接口。汤匙、刀和叉子都实现了餐具接口。

抽象工厂的接口定义如下所示；

[html] [view plaincopyprint?](#)

```
1. package abstractFactory;
2.
3. public interface KitchenFactory{
4.
5.     public Food getFood();
6.
7.     public TableWare getTableWare();
8. }
```


9. }

10.

11.抽象餐具的接口定义如下所示:

12.

13.package abstractFactory;

14.

15.public interface TableWare{

16.

17. public String getTool();

18.

19.}

20.

21.抽象事物的接口定义如下所示:

22.

23.package abstractFactory;

24.

25.public interface Food{

26.

27. public String getEatable();

28.

29.}

30.

31.而具体的实现也非常简单,以 AKitchen 为例子

32.

33.具体工厂 AKitchen 的定义如下所示;

34.

35.package abstractFactory;

36.

37.public class AKitchenimplements KitchenFactory{

38.

39. public Food getFood(){

40.

41. return new Milk();

42.

43. }

```
44.  
45. public TableWare getTableWare(){  
46.  
47.     return new Spoon();  
48.  
49. }  
50.  
51.}  
52.  
53.  
54.  
55.具体餐具(spoon)的定义如下所示:  
56.  
57.package abstractFactory;  
58.  
59.public class Spoonimplements TableWare{  
60.  
61.    public String getTool() {  
62.  
63.        return "spoon";  
64.  
65.    }  
66.  
67.}  
68.  
69.具体食物(milk)的定义如下所示:  
70.  
71.package abstractFactory;  
72.  
73.public class Milkimplements Food{  
74.  
75.    public String getEatable(){  
76.  
77.        return "milk";  
78.
```

```
79. }
80.
81.}
82.
83.客户端的定义如下:
84.
85.package abstractFactory;
86.
87.public class Client{
88.
89.    public void eat(KitchenFactory k){
90.
91.        System.out.println("A person eat "+k.getFood().getEatable()
92.
93.            +" with "+k.getTableWare().getTool()+"!");
94.
95.    }
96.
97.    public static void main(String[] args){
98.
99.        Client client=    Client();
100.
101.        KitchenFactory kf =    AKitchen();
102.
103.        client.eat(kf);
104.
105.        kf=    BKitchen();
106.
107.        client.eat(kf);
108.
109.        kf=    CKitchen();
110.
111.        client.eat(kf);
112.
113.    }
```

114.

115. }

在以下情况下应当考虑使用抽象工厂模式：

- 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。
- 这个系统有多于一个的产品族，而系统只消费其中某一产品族。
- 同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。
- 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

（三十三）大话设计模式（三）单例模式

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

单例模式属于对象创建型模式，其意图是保证一个类仅有一个实例，并提供一个访问它的全局访问点。对一些类来说，只有一个实例是很重要的，虽然系统中可以有許多打印机，但却只应该有一个打印机假脱机，只应该有一个文件系统和一个窗口管理器，一个数字滤波器只能有一个 A/D 转换器，一个会计系统只能专用于一个公司。怎样才能保证一个类只有一个实例并且这个实例易于被访问，一个全局变量使得一个对象可以被访问，但它不能防止你实例化多个对象，一个更好的方法是让类自身负责保存他的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该实例的方法，这就是 Singleton 模式。

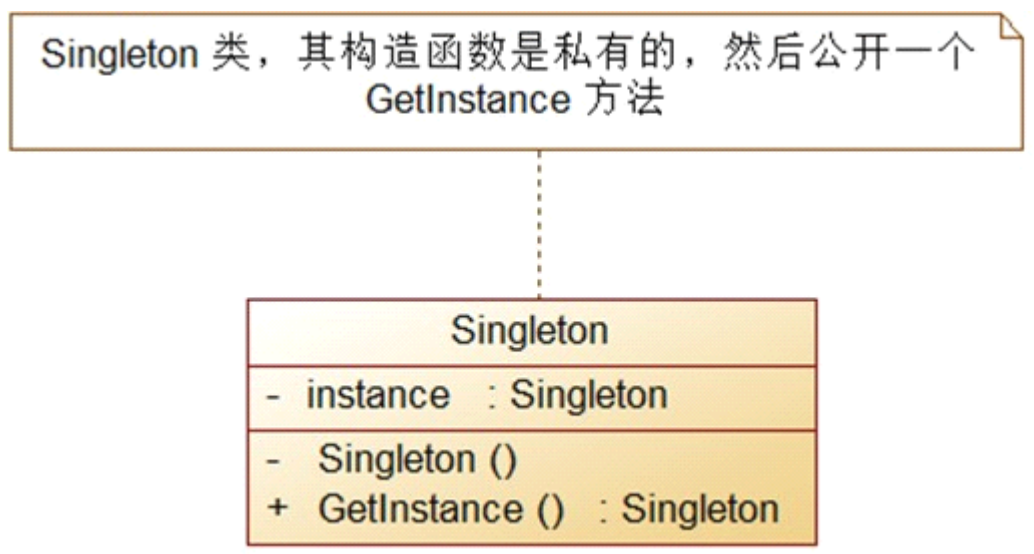
单例模式的要点

- 一是某个类只能有一个实例；
- 二是它必须自行创建这个实例；
- 三是它必须自行向整个系统提供这个实例。

实用性：在下面的情况下可以使用 Singleton 模式。

- I 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
- I 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

类图：



从上面的类图中可以看出，在单例类中有一个构造函数 Singleton，但是这个构造函数却是私有的（前面是“-”符号），然后在里面还公开了一个 GetInstance（）方法，通过上面的类图不难看出单例模式的特点，从而也可以给出单例模式的定义：
单例模式保证一个类仅有一个实例，同时这个类还必须提供一个访问该类的全局访问点。

下面来自：<http://blog.csdn.net/zhengzhib/article/details/7331369>，博主给出的单例模式的详解不得不学习

单例模式根据实例化对象时机的不同分为两种：一种是饿汉式单例，一种是懒汉式单例。饿汉式单例在单例类被加载时候，就实例化一个对象交给自己的引用；而懒汉式在调用取得实例方法的时候才会实例化对象。代码如下：

饿汉式单例

[\[html\] view plaincopyprint?](#)

1. public class Singleton {
- 2.

```

3.   private static Singleton singleton = Singleton();
4.
5.   private Singleton(){}
6.
7.   public static Singleton getInstance(){
8.       return singleton;
9.
10.  }
11.
12.}

```

懒汉式单例

[html] [view plaincopyprint?](#)

```

1. public class Singleton {
2.
3.     private static Singleton singleton;
4.
5.     private Singleton(){}
6.
7.     public static synchronized Singleton getInstance(){
8.
9.         if(singleton==null){
10.
11.             singleton = Singleton();
12.
13.         }
14.
15.         return singleton;
16.     }
17.
18.}

```

单例模式的优点：

- 在内存中只有一个对象，节省内存空间。
- 避免频繁的创建销毁对象，可以提高性能。
- 避免对共享资源的多重占用。
- 可以全局访问。

适用场景： 由于单例模式的以上优点，所以是编程中用的比较多的一种设计模式。我总结了一下我所知道的适合使用单例模式的场景：

- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
- 有状态的工具类对象。
- 频繁访问数据库或文件的对象。
- 以及其他我没用过的所有要求只有一个对象的场景。

单例模式注意事项：

- 只能使用单例类提供的方法得到单例对象，不要使用反射，否则将会实例化一个新对象。
- 不要做断开单例类对象与类中静态引用的危险操作。
- 多线程使用单例使用共享资源时，注意线程安全问题。

关于 java 中单例模式的一些争议：

单例模式的对象长时间不用会被 jvm 垃圾收集器收集吗

看到不少资料中说：如果一个单例对象在内存中长久不用，会被 jvm 认为是一个垃圾，在执行垃圾收集的时候会被清理掉。对此这个说法，笔者持怀疑态度，笔者本人的观点是：在 hotspot 虚拟机 1.6 版本中，除非人为地

断开单例中静态引用到单例对象的联接，否则 **jvm** 垃圾收集器是不会回收单例对象的。

对于这个争议，笔者单独写了一篇文章进行讨论，如果您有不同的观点或者有过这方面的经历请进入文章[单例模式讨论篇：单例模式与垃圾收集参与讨论](#)。

在一个 **jvm** 中会出现多个单例吗

在分布式系统、多个类加载器、以及序列化的的情况下，会产生多个单例，这一点是无庸置疑的。那么在同一个 **jvm** 中，会不会产生单例呢？使用单例提供的 `getInstance()` 方法只能得到同一个单例，除非是使用反射方式，将会得到新的单例。代码如下

```
Class c = Class.forName(Singleton.class.getName());
Constructor ct = c.getDeclaredConstructor();
ct.setAccessible(true);
Singleton singleton = (Singleton)ct.newInstance();
```

这样，每次运行都会产生新的单例对象。所以运用单例模式时，一定要注意不要使用反射产生新的单例对象。

懒汉式单例线程安全吗

主要是网上的一些说法，懒汉式的单例模式是线程不安全的，即使是在实例化对象的方法上加 `synchronized` 关键字，也依然是危险的，但是笔者经过编码测试，发现加 `synchronized` 关键字修饰后，虽然对性能有部分影响，但是却是线程安全的，并不会产生实例化多个对象的情况。

单例模式只有饿汉式和懒汉式两种吗

饿汉式单例和懒汉式单例只是两种比较主流和常用的单例模式方法，从理论上讲，任何可以实现一个类只有一个实例的设计模式，都可以称为单例模式。

单例类可以被继承吗

饿汉式单例和懒汉式单例由于构造方法是 `private` 的，所以他们都是不可继承的，但是其他很多单例模式是可以继承的，例如登记式单例。

饿汉式单例好还是懒汉式单例好

在 `java` 中，饿汉式单例要优于懒汉式单例。`C++` 中则一般使用懒汉式单例。

单例模式比较简单，在此就不举例代码演示了。

（三十四）大话设计模式（四）策略模式

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

Strategy 是属于设计模式中 对象行为型模式，主要是定义一系列的算法，把这些算法一个个封装成单独的类。

定义：策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。（原文：The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.）

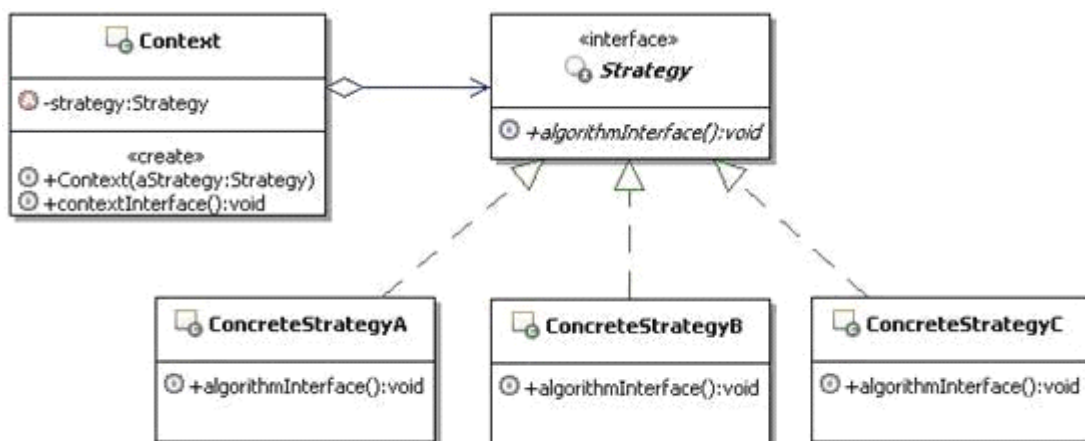
角色：

Strategy：策略接口，用来约束一系列具体的策略算法。Context 使用这个接口来调用具体的策略实现定义的算法。

ConcreteStrategy：具体的策略实现，也就是具体的算法实现。

Context：上下文，负责和具体的策略类交互，通常上下文会持有一个真正的策略实现，上下文还可以让具体的策略类来获取上下文的数据，甚至让具体的策略类来回调上下文的方法。

策略模式的结构示意图如图 1 所示：



应用场景:

- 1、 多个类只区别在表现行为不同, 可以使用 **Strategy** 模式, 在运行时动态选择具体要执行的行为。
- 2、 需要在不同情况下使用不同的策略(算法), 或者策略还可能在未来用其它方式来实现。
- 3、 对客户隐藏具体策略(算法)的实现细节, 彼此完全独立。

优点:

- 1、 简化了单元测试, 因为每个算法都有自己的类, 可以通过自己的接口单独测试。
- 2、 避免程序中使用多重条件转移语句, 使系统更灵活, 并易于扩展。
- 3、 遵守大部分 **GRASP** 原则和常用设计原则, 高内聚、低耦合。
4. 提供了一种替代继承的方法, 而且既保持了继承的优点(代码重用)还比继承更灵活(算法独立, 可以任意扩展)。

缺点:

- 1、 因为每个具体策略类都会产生一个新类, 所以会增加系统需要维护的类的数量。

2、 在基本的策略模式中，选择所用具体实现的职责由客户端对象承担，并转给策略模式的 **Context** 对象。（这本身没有解除客户端需要选择判断的压力，而策略 模式与简单工厂模式结合后，选择具体实现的职责也可以由 **Context** 来承担，这就最大化的减轻了客户端的压力。）

举一个我看过的策略模式比较经典的例子：报价管理

对不同的客户要报不同的价格，向客户报价是非常复杂的，因此在一些 CRM（客户关系管理）的系统中，会有一个单独的报价管理模块，来处理复杂的报价功能。

为了演示的简洁性，假定现在需要实现一个简化的报价管理，实现如下的功能：

- （1）对普通客户或者是新客户报全价
- （2）对老客户报的价格，统一折扣 5%
- （3）对客户报的价格，统一折扣 10%

（1）先看策略接口，示例代码如下：

[java] [view plaincopyprint?](#)

```
1. /**
2.
3.  * 策略，定义计算报价算法的接口
4.
5.  */
6.
7. public interface Strategy {
8.
9.     /**
10.
```

```

11.    * 计算应报的价格
12.
13.    * @param goodsPrice 商品销售原价
14.
15.    * @return 计算出来的，应该给客户报的价格
16.
17.    */
18.
19.    public double calcPrice(double goodsPrice);
20.
21. }

```

(2) 接下来看看具体的算法实现，不同的算法，实现也不一样，先看为新客
户或者是普通客户计算应报的价格的实现，示例代码如下：

[java] [view plaincopyprint?](#)

```

1.  /**
2.
3.    * 具体算法实现，为新客户或者是普通客户计算应报的价格
4.
5.    */
6.
7.    public class NormalCustomerStrategy implements Strategy{
8.
9.        public double calcPrice(double goodsPrice) {
10.
11.            System.out.println("对于新客户或者是普通客户，没有折扣");
12.
13.            return goodsPrice;
14.
15.        }
16.
17.    }

```

再看看为老客户计算应报的价格的实现，示例代码如下：

[java] [view plaincopyprint?](#)

```
1.  /**
2.
3.   * 具体算法实现，为老客户计算应报的价格
4.
5.   */
6.
7.  public class OldCustomerStrategy implements Strategy{
8.
9.      public double calcPrice(double goodsPrice) {
10.
11.          System.out.println("对于老客户，统一折扣 5%");
12.
13.          return goodsPrice*(1-0.05);
14.
15.      }
16.
17. }
```

再看看为大客户计算应报的价格的实现，示例代码如下：

[java] [view plaincopyprint?](#)

```
1.  /**
2.
3.   * 具体算法实现，为大客户计算应报的价格
4.
5.   */
6.
7.  public class LargeCustomerStrategy implements Strategy{
8.
```



```

9.     public double calcPrice(double goodsPrice) {
10.
11.         System.out.println("对于大客户，统一折扣 10%");
12.
13.         return goodsPrice*(1-0.1);
14.
15.     }
16.
17. }

```

(3) 接下来看看上下文的实现，也就是原来的价格类，它的变化比较大，主要有：

- 原来那些私有的，用来做不同计算的方法，已经去掉了，独立出去做成了算法类
- 原来报价方法里面，对具体计算方式的判断，去掉了，让客户端来完成选择具体算法的功能
- 新添加持有一个具体的算法实现，通过构造方法传入
- 原来报价方法的实现，变化成了转调具体算法来实现

示例代码如下：

[java] [view plaincopyprint?](#)

```

1.  /**
2.
3.     * 价格管理，主要完成计算向客户所报价格的功能
4.
5.     */
6.

```

```
7. public class Price {
8.
9.     /**
10.
11.     * 持有一个具体的策略对象
12.
13.     */
14.
15.     private Strategy strategy = null;
16.
17.     /**
18.
19.     * 构造方法，传入一个具体的策略对象
20.
21.     * @param aStrategy 具体的策略对象
22.
23.     */
24.
25.     public Price(Strategy aStrategy){
26.
27.         this.strategy = aStrategy;
28.
29.     }
30.
31.     /**
32.
33.     * 报价，计算对客户的报价
34.
35.     * @param goodsPrice 商品销售原价
36.
37.     * @return 计算出来的，应该给客户报的价格
38.
39.     */
40.
41.     public double quote(double goodsPrice){
```

```
42.  
43.     return this.strategy.calcPrice(goodsPrice);  
44.  
45. }  
46.  
47. }
```

(4) 写个客户端来测试运行一下，好加深体会，示例代码如下：

[html] [view plaincopyprint?](#)

```
1.  public class Client {  
2.  
3.     public static void main(String[] args) {  
4.  
5.         //1: 选择并创建需要使用的策略对象  
6.  
7.         Strategy strategy =     LargeCustomerStrategy ();  
8.  
9.         //2: 创建上下文  
10.  
11.        Price ctx =     Price(strategy);  
12.  
13.        //3: 计算报价  
14.  
15.        double quote =     .quote(1000);  
16.  
17.        System.out.println("向客户报价: "+quote);  
18.  
19.    }  
20.  
21. }
```

（三十四）大话设计模式（五）创建者模式和原型模式

本文来自：曹胜欢博客专栏。转载请注明出处：

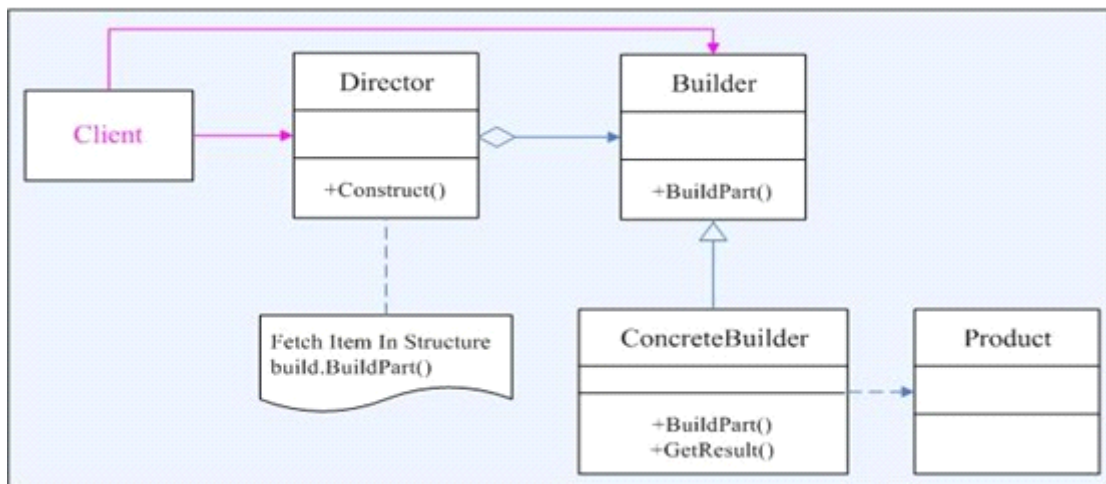
<http://blog.csdn.net/csh624366188>

创建者模式是创建型模式中最负责的一个设计模式了，创建者负责构建一个对象的各个部分，并且完成组装的过程。构建模式主要用来针对复杂产品生产，分离部件构建细节，以达到良好的伸缩性。把构造对象实例的逻辑移到了类的外部，在这个类外部定义了这个类的构造逻辑。它把一个复杂对象的构造过程从对象的表示中分离出来。其直接效果是将一个复杂的对象简化为一个比较简单的对象。它强调的是产品的构造过程。

在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法确相对稳定。如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的“稳定构建算法”不随着需求改变而改变？这就是要说的建造者模式。

意图：将一个复杂对象的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

类图：



抽象建造者（Builder）： 给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

具体建造者（Concrete Builder）： 实现 Builder 接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。

指导者（Director）： 调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

产品（Product）： 要创建的复杂对象。

适用范围：

- 1.需要生成的产品对象有复杂的内部结构。
- 2.需要生成的产品对象的属性相互依赖，建造者模式可以强迫生成顺序。
- 3.在对象创建过程中会使用到系统中的一些其他对象，这些对象在产品对象的创建过程中不易得到。

效果：

- 1.建造者模式的使用时的产品的内部表象可以独立的变化。使用建造者模式可以使客户端不必知道产品内部组成的细节。
- 2.每一个 Builder 都相对独立，而与其他 Builder 无关。
- 3.模式所建造的最终产品易于控制。

下面我们来看一下经典创建者模式的一个实现形式，然后针对这个经典模式后面提出几个改进方案，我们这里以上面讲述的服装的过程作为例子来说明下创建者模式的原理和思想，希望大家也能灵活的运用到实际的项目中去。达到学以致用目的。

我们来看看具体的代码实现：

[html] [view plaincopyprint?](#)

```
1. package builder;
2.
3. public class Product {
4.
5.     ArrayList<String> parts =      ArrayList<String>();
6.
7.     public void add(String part)
8.
9.     {
10.
11. parts.add(part);
12.
13. }
14.
15. public void show()
16.
17. {
18.
19. System.out.println("产品创建-----");
```

```
20.  
21.for(String part : parts)  
22.  
23.{  
24.  
25.System.out.println(part);  
26.  
27.}  
28.  
29.}  
30.  
31.}  
32.  
33.public abstract class Builder {  
34.  
35.public abstract void BuildPartA();  
36.  
37.public abstract void BuildPartB();  
38.  
39.public abstract Product getResult();  
40.  
41.}  
42.  
43.public class ConcreteBuilder1 extends Builder{  
44.  
45.private Product product = new Product();  
46.  
47.@Override  
48.  
49.public void BuildPartA() {  
50.  
51.product.add("部件 A");  
52.  
53.}  
54.
```

```
55.@Override
56.
57.public void BuildPartB() {
58.
59.product.add("部件 B");
60.
61.}
62.
63.@Override
64.
65.public Product getResult() {
66.
67.return product;
68.
69.}
70.
71.}
72.
73.public class ConcreteBuilder2 extends Builder{
74.
75.private Product product =    Product();
76.
77.@Override
78.
79.public void BuildPartA() {
80.
81.product.add("部件 x");
82.
83.}
84.
85.@Override
86.
87.public void BuildPartB() {
88.
89.product.add("部件 y");
```



```
90.
91.}
92.
93.@Override
94.
95.public Product getResult() {
96.
97.return product;
98.
99.}
100.
101. }
102.
103. public class Director {
104.
105. public void Construct(Builder builder)
106.
107. {
108.
109. builder.BuildPartA();
110.
111. builder.BuildPartB();
112.
113. }
114.
115. }
116.
117. public class TestBuilder {
118.
119. public static void main(String[] args) {
120.
121. Director director = new Director();
122.
123. Builder b1 = new ConcreteBuilder1();
124.
```

```
125.  Builder b2 =      ConcreteBuilder2();
126.
127.  director.Construct(b1);
128.
129.  Product p1 =      .getResult();
130.
131.  p1.show();
132.
133.  director.Construct(b2);
134.
135.  Product p2 =      .getResult();
136.
137.  p2.show();
138.
139.  }
140.
141.  }
```

通过上面的代码，我们给出了经典创建者模式的核心代码形式，那么针对上面无疑有以下的几个缺点：

- 1、Ibuilder 接口必须定义完整的组装流程，一旦定义就不能随意的动态修改。
- 2、Builder 与具体的对象之间有一定的依赖关系，当然这里可以通过接口来解耦来实现灵活性。
- 3、Builder 必须知道具体的流程。

那么针对上面的几个问题，我们如何来解决呢？我想前面的创建型模式已经给我了足够的经验，还是通过配置文件或者其他的形式来提供灵活性。

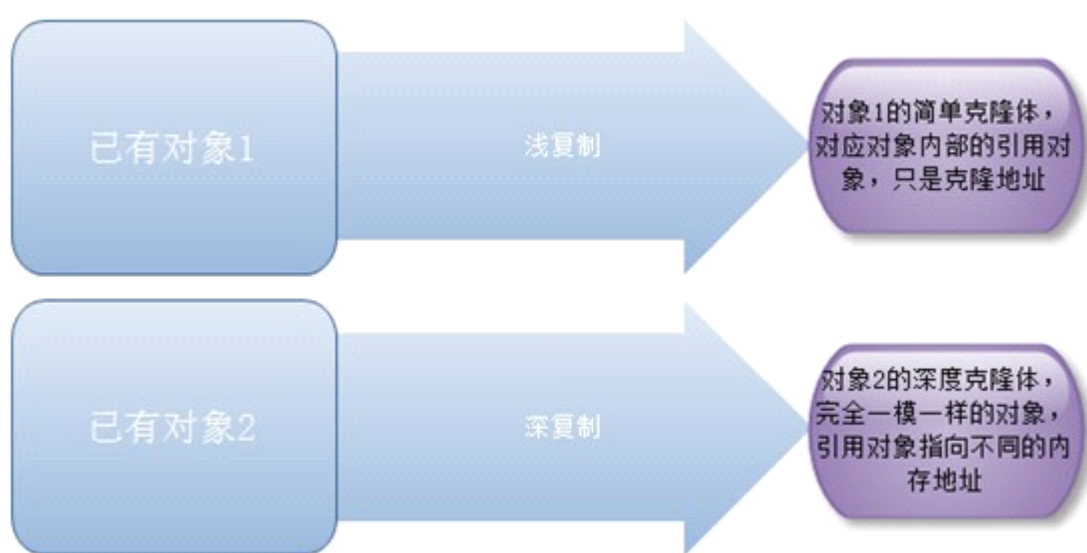
原型模式

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。Prototype 原型模式是一种创建型设计模式，Prototype 模式允许一个对象

再创建另外一个可定制的对象，根本无需知道任何如何创建的细节,工作原理是:通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。它主要面对的问题是：“某些结构复杂的对象”的创建工作；由于需求的变化，这些对象经常面临着剧烈的变化，但是他们却拥有比较稳定一致的接口。

原型模式最大的特点是克隆一个现有的对象，这个克隆的结果有 2 种，**一种是浅复制，另一种是深复制**，这里我们也会探讨下深复制和浅复制的原理，这样可能更方便大家理解这个原型模式的使用。我们都知道，创建型模式一般是用来创建一个新的对象，然后我们使用这个对象完成一些对象的操作，我们通过原型模式可以快速的创建一个对象而不需要提供专门的 new() 操作就可以快速完成对象的创建，这无疑是一种非常有效的方式，快速的创建一个新的对象。

原型模式的原理图：



原型模式的主要思想是**基于现有的对象克隆一个新的对象出来**，一般是有对象的内部提供克隆的方法，通过该方法返回一个对象的副本，这种创建对象

的方式，相比我们之前说的几类创建型模式还是有区别的，之前的讲述的工厂模式与抽象工厂都是通过工厂封装具体的 `new` 操作的过程，返回一个新的对象，有的时候我们通过这样的创建工厂创建对象不值得，特别是以下的几个场景的时候，可能使用原型模式更简单也效率更高。

主要运用场合：

- 1、如果说我们的对象类型不是刚开始就能确定，而是这个类型是在运行期确定的话，那么我们通过这个类型的对象克隆出一个新的类型更容易。
- 2、有的时候我们可能在实际的项目中需要一个对象在某个状态下的副本，这个前提很重要，这点怎么理解呢，例如有的时候我们需要对比一个对象经过处理后的状态和处理前的状态是否发生过改变，可能我们就需要在执行某段处理之前，克隆这个对象此时状态的副本，然后等执行后的状态进行相应的对比，这样的应用在项目中也是经常会出现的。
- 3、当我们在处理一些对象比较简单，并且对象之间的区别很小，可能只是很固定的几个属性不同的时候，可能我们使用原型模式更合适

深复制和浅复制：

(1)浅复制（浅克隆）

被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

(2)深复制（深克隆）

被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原

有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍。

下面举一个实例来实现以下原型模式：

[html] [view plaincopyprint?](#)

```
1. import java.io.ByteArrayInputStream;
2. import java.io.ByteArrayOutputStream;
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5. import java.io.ObjectOutputStream;
6. import java.io.Serializable;
7.
8. class Prototype implements Cloneable,Serializable{
9.
10. private String str;
11. private Temp temp;
12.
13. public Object clone()throws CloneNotSupportedException{ //浅克隆
14. Prototype prototype=(Prototype)super.clone();
15. return prototype;
16. }
17. public Object deepClone()throws IOException,ClassNotFoundException{ //深克隆
18. ByteArrayOutputStream bo=    ByteArrayOutputStream();
19. ObjectOutputStream oo=    ObjectOutputStream(bo);
20. oo.writeObject(this);
21.
22. ByteArrayInputStream bi=    ByteArrayInputStream(bo.toByteArray());
23. ObjectInputStream oi=    ObjectInputStream(bi);
24. return oi.readObject();
25. }
26. public String getStr() {
27. return str;
28. }
```

```

29. public void setStr(String str) {
30.     this.str =    ;
31. }
32. public Temp getTemp() {
33.     return temp;
34. }
35. public void setTemp(Temp temp) {
36.     this.temp =    ;
37. }
38.
39.}
40.class Temp implements Serializable{
41.
42.}
43.public class Test {
44.
45. public static void main(String[] args)throws CloneNotSupportedException,ClassN
        otFoundException ,IOException{
46.
47.     Prototype pt=    Prototype();
48.     Temp temp=    Temp();
49.     pt.setTemp(temp);
50.     pt.setStr("Hello World");
51.     System.out.println("使用浅克隆方法进行创建对象");
52.     Prototype pt1=(Prototype)pt.clone();
53.     System.out.println("=====");
54.     System.out.println("比较 pt 和 pt1 的 str 的值: ");
55.     System.out.println(pt.getStr());
56.     System.out.println(pt1.getStr());
57.
58.     System.out.println("修改 pt1 对象中 str 的值后，比较 pt 和 pt1 的 str 的值: ");
59.     pt1.setStr("你好，世界");
60.     System.out.println(pt.getStr());
61.     System.out.println(pt1.getStr());
62.     System.out.println("=====");

```

```

63. System.out.println("比较 pt 和 pt1 中 temp 对象的值");
64. System.out.println(pt.getTemp());
65. System.out.println(pt1.getTemp());
66.
67. System.out.println("使用深克隆方法进行创建对象");
68. System.out.println("=====");
69. pt1=(Prototype)pt.deepClone();
70. System.out.println(pt.getTemp());
71. System.out.println(pt1.getTemp());
72.
73.
74.
75. }
76.
77.}

```

输出结果:

使用浅克隆方法进行创建对象

=====

比较 pt 和 pt1 的 str 的值:

Hello World

Hello World

修改 pt1 对象中 str 的值后, 比较 pt 和 pt1 的 str 的值:

Hello World

你好, 世界

=====

比较 pt 和 pt1 中 temp 对象的值

Temp@affc70

Temp@affc70

使用深克隆方法进行创建对象

=====

Temp@affc70

Temp@15d56d5

从上面的输出结果我们可以看出使用 `Object.clone()` 方法只能浅层次的克隆，即只能对那些成员变量是基本类型或 `String` 类型的对象进行克隆，对那些成员变量是类类型的对象进行克隆要使用到对象的序列化，不然克隆克隆出来的 `Prototype` 对象都共享同一个 `temp` 实例。

总结：

原型模式作为创建型模式中的最特殊的一个模式，具体的创建过程，是由对象本身提供，这样我们在很多的场景下，我们可以很方便的快速的构建新的对象，就像前面分析讲解的几类场景中，可能我们通过使用对象的克隆，比通过其他几类的创建型模式，效果要好的多，而且代价也小很多。打个比方，

原型模式对于系统的扩展，可以做到无缝的扩展，为什么这么说呢？比如其他的创建型工厂，如果新增一个对象类型，那么我们不管是修改配置文件的方式，还是修改代码的形式，无疑我们都是需要进行修改的，对于我们大家通用的公共应用来说这无疑是非常危险的，那么通过原型模式，则可以解决这样的问题，

因为类型本身实现这样的方法即可，但是也有一定的缺点，每个对象都实现这样的方法，无疑是很大的工作量，但是在某些特殊的环境下，或者实际的项目中，可能原型模式是好的选择。

（三十五）细谈 struts2(一) 自己实现 struts2 框架

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

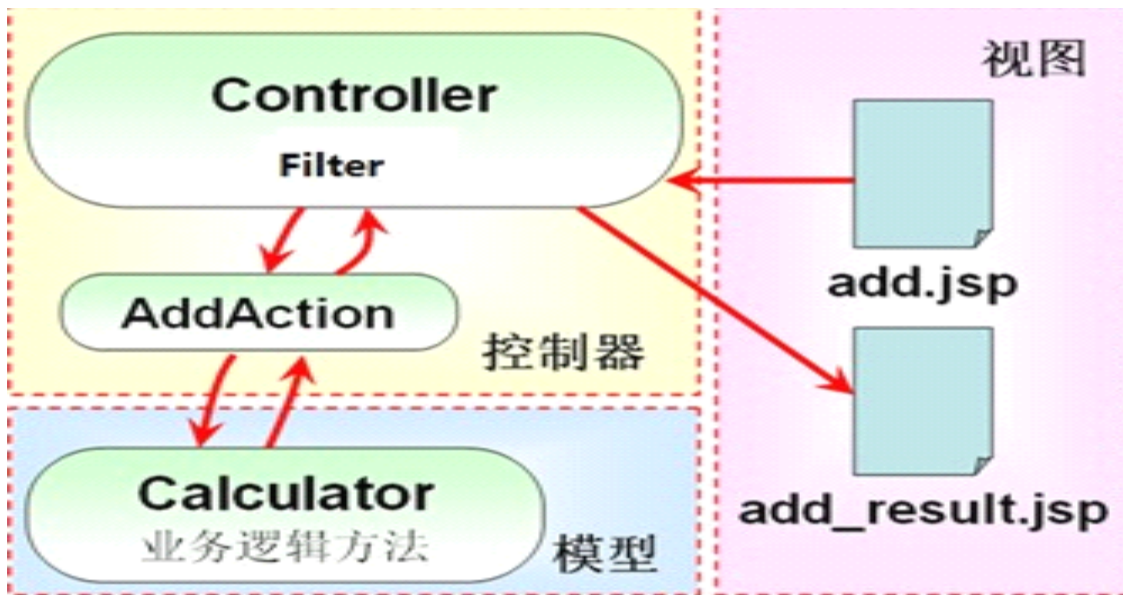
Struts 最早是作为 Apache Jakarta 项目的组成部分,项目的创立者希望通过对该项目的研究,改进和提高 JavaServer Pages、[Servlet](#)、[标签库](#)以及面向对象的技术水准。最初的 **struts1.x** 很快在企业开发中流行了起来,与此同时,当时还有一个非常的优秀的 web 开发框架诞生,那就是 **webwork**,但 **webwork** 没有像 **struts1** 那么幸运,没有得到流行,但 **webwork** 简洁、灵活功能强大等优点绝不输于当时流行的 **strut1.x**。当然 **struts1** 开发人员不是也没有意识到这一点。于是 **struts** 和 **WebWork** 得到了结合,**webwork** 算是利用 **struts** 的名气来发展自己吧,于是 **struts2** 诞生了。

Struts2 概述

Struts 2 是 **Struts** 的下一代产品,是在 **struts** 和 **WebWork** 的技术基础上进行了合并的全新的 **Struts 2** 框架。其全新的 **Struts 2** 的体系结构与 **Struts 1** 的体系结构的差别巨大。**Struts 2** 以 **WebWork** 为核心,采用拦截器的机制来处理用户的请求,这样的设计也使得业务逻辑控制器能够与 **Servlet API** 完全脱离开,所以 **Struts 2** 可以理解为 **WebWork** 的更新产品。虽然从 **Struts 1** 到 **Struts 2** 有着太大的变化,但是相对于 **WebWork**,**Struts 2** 只有很小的变化。由于 **struts1** 现在开发中很少在用到,所以我们直接进入 **struts2** 的学习,但

以前的项目中还是大多数保留着 struts1 的应用。由于 struts 是基于 mvc 模式的框架，所以我们学习 struts 的第一步就是开发自己的基于 MVC 的框架

首先看一下一个 **MVC** 的流程图的例子：



就像图中例子，在视图层 add.jsp 中写一个提交两个数据的表单，表单提交给控制器，在控制器中通过它所提交的 uri 获得表单所要提交的 action，然后把请求交给 action，然后在 action 中调用业务逻辑的方法进行逻辑运算，获得结果，把结果保存起来，然后，把所有返回的界面作为返回结果返回给控制器，然后控制器根据返回的界面的字符串选择转发到该界面

下面我们就用程序，把这个流程实现出来：

1. 首先要把表单界面写出来：add.jsp

[html] view plaincopyprint?

1. `<form action=` `method=` `><div align=` `><font color=`
`>`
- 2.
3. ``加法器实现
`
`
- 4.

```

5.     </div><table align="center">
6.
7.     <tr>
8.
9.     <td>第一个数: </td>
10.
11.    <td><input type="text" name="num1" /></td>
12.
13.    </tr>
14.
15.    <tr>
16.
17.    <td>第二个数: </td>
18.
19.    <td><input type="text" name="num2" /></td>
20.
21.    </tr>
22.
23.    <tr align="center">
24.
25.    <td colspan="2"><input type="button" value="求和" /> <input type="button" value="求差" /></td>
26.
27.    </tr>
28.
29.    </table>
30.
31. </form>

```

2.创建控制器，其实这里的控制器就是一个 servlet，这里我们给规定凡是请求后缀是.action 的都提交到这个控制器里，controller.java:

[html] [view plain](#)copyprint?

```

1. public void doPost(HttpServletRequest request, HttpServletResponse response)

```

```

2.
3. throws ServletException, IOException {
4.
5. String path=      .getRequestURI();
6.
7. String realPath=   .substring(path.lastIndexOf("/") + 1, path.lastIndexOf("."));
8.
9. Action action=    ;
10.
11. String path2=     ;
12.
13. if("add".equals(realPath)){
14.
15. action=      AddAction();
16.
17. path2=       .execute(request, response);
18.
19. }
20.
21.      .....
22.
23.      If(...){
24.
25.          .....
26.
27.      }
28.
29. request.getRequestDispatcher(path2).forward(request, response);
30.
31. }

```

因为控制器是一个 servlet，所以在 web.xml 中要对他进行配置：

[html] [view plaincopyprint?](#)

```
1. <?xml version=      encoding=      ?>
2.
3. <web-app version=
4.
5. xmlns=
6.
7. xmlns:xsi=
8.
9. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
10.
11.http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
12.
13. <servlet>
14.
15.   <servlet-name>Controller</servlet-name>
16.
17.   <servlet-class>zxj.struts2.servlet.Controller</servlet-class>
18.
19. </servlet>
20.
21. <servlet-mapping>
22.
23.   <servlet-name>Controller</servlet-name>
24.
25.   <url-pattern>*.action</url-pattern>
26.
27. </servlet-mapping>
28.
29. </web-app>
```

下面来看一下 action 里面应该写的内容，由于一直以来都提倡面向接口编程，并且面向接口编程也能很好的体现 java 的可扩展性，所以我们对所有的 action 提供一个共同的接口：action.java:

[html] view plaincopyprint?

```
1. public interface Action {  
2.  
3. public String result(HttpServletRequest request, HttpServletResponse response);  
4.  
5. }
```

下面是具体的 action 实现：addaction.java：其中具体的业务逻辑调用的 add 方法就是两个数相加，这里就不贴代码了：

[java] view plaincopyprint?

```
1. public String execute(HttpServletRequest request,  
2.  
3. HttpServletResponse response) {  
4.  
5. double i=Double.parseDouble(request.getAttribute("firstNmb").toString());  
6.  
7. double n=Double.parseDouble(request.getAttribute("secondNmb").toString());  
8.  
9. Calculator c=new Calculator();  
10.  
11.double result=c.add(i, n);  
12.  
13.request.setAttribute("result", result);  
14.  
15.return "add_result.jsp";  
16.  
17.}  
18.  
19.}
```

这些就是我们自己写的 mvc 的基本框架，当然这里面有很多不足的地方，这里只是为了演示基于 mvc 框架的基本架构，具体细节都可以细化和扩展性的实现，比如控制器里面的选择哪个 action，**这个可以用配置文件来实现的，基本思路：在控制器中获得所请求 action 的前缀名，然后去解析所配置的文件，在然后拿着这个前缀名去找配置文件中相符的 action 所对应的类，然后在利用反射执行对应类的方法，根据然后在执行完 action 后，获得结果，然后从配置中获得该结果对应的界面**，这样就可以很好的体现了这个程序的可扩展性了。

到这里我相信大家应该对基于 mvc 的框架的执行流程有一定的了解了，相信大家一定对学习 struts2 框架迫不及待了，那大家就等待着下一篇博客：**细谈 struts2 之初识 struts2 框架**

（三十六）大话设计模式（六）观察者模式

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

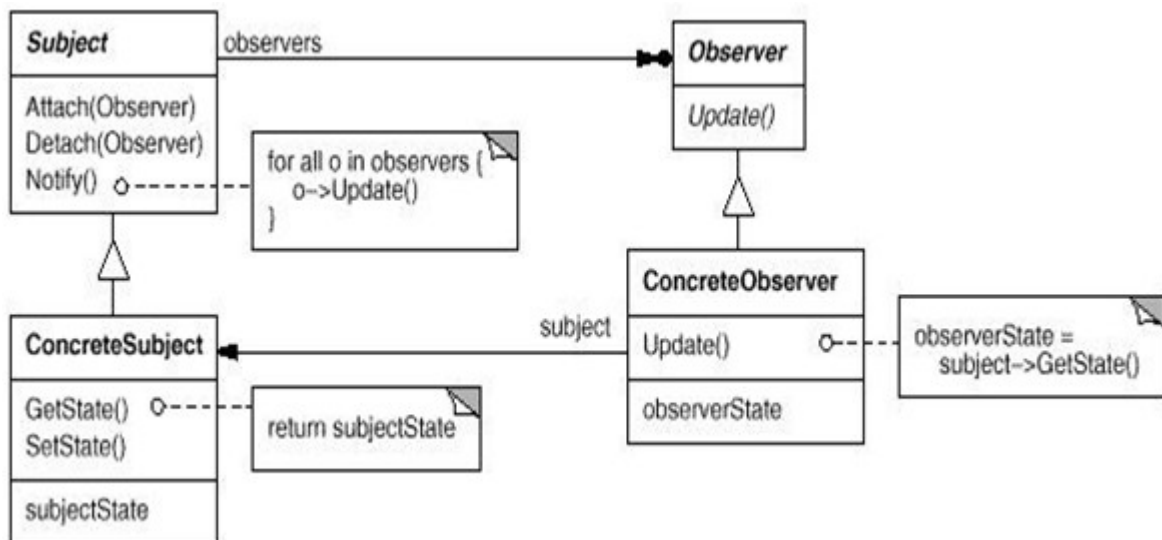
Observer 模式是行为模式之一，它的作用是当一个对象的状态发生变化时，能够自动通知其他关联对象，自动刷新对象状态。Observer 模式提供给关联对象一种同步通信的手段，使某个对象与依赖它的其他对象之间保持状态同步。《设计模式》一书对 Observer 是这样描述的：**定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知并自动更新**。别名：依赖(Dependents)、发布-订阅(Publish-Subscribe)。

下面我们就来看看观察者模式的组成部分。

- 1) 抽象目标角色 (Subject)：目标角色知道它的观察者，可以有任意多个观察者观察同一个目标。并且提供注册和删除观察者对象的接口。目标角色往往由抽象类或者接口来实现。
- 2) 抽象观察者角色 (Observer)：为那些在目标发生改变时需要获得通知的对象定义一个更新接口。抽象观察者角色主要由抽象类或者接口来实现。
- 3) 具体目标角色 (Concrete Subject)：将有关状态存入各个 Concrete Observer 对象。当它的状态发生改变时，向它的各个观察者发出通知。

4) 具体观察者角色（Concrete Observer）：存储有关状态，这些状态应与目标的状态保持一致。实现 Observer 的更新接口以使自身状态与目标的状态保持一致。在本角色内也可以维护一个指向 Concrete Subject 对象的引用。

结构如下：



观察者模式的优缺点：

观察者模式的效果有以下的优点：

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知，

观察者模式有下面的缺点：

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

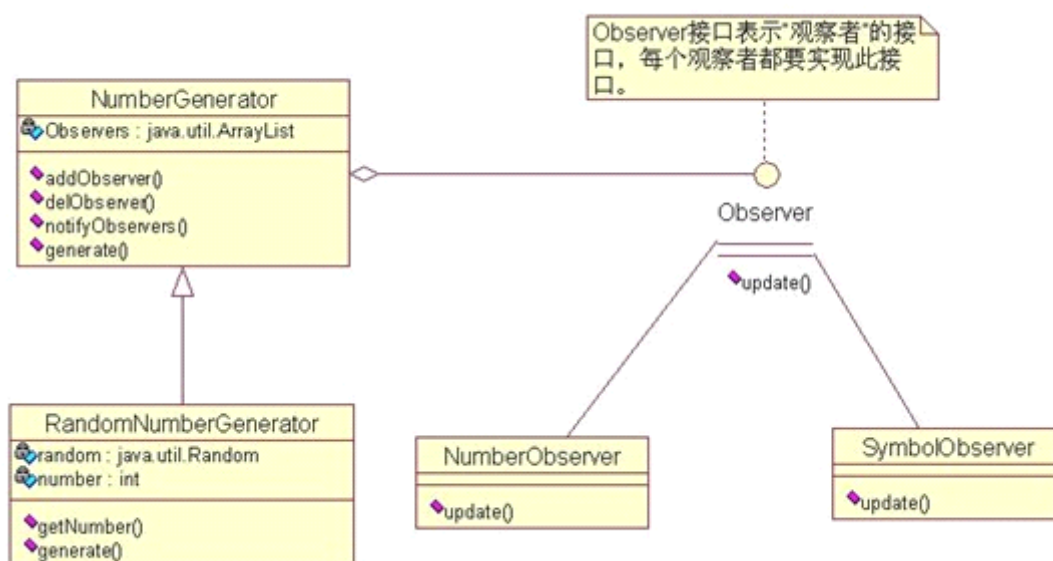
第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

下面用程序来说明一下观察者模式的流程：

一个随机数产生对象和两个观察者，这两个观察者都在随机数产生对象那里注册了，意思说如果你产生了新的数字，就通知我一声。

结构图：



类说明

名称

功能说明

Observer

表示观察者的接口，要成为观察者必须实现此接口才行

NumberGenerator

表示产生数值的抽象类

RandomNumberGenerator

产生随机数的类，继承于
NumberGenerator

NumberObserver

数字观察者，会打印出变化的数字

SymbolObserver

符号观察者,打印 *N* 个符号,打印多少个符号，由接受到的数值确定

1.Observer

[java] [view plaincopyprint?](#)

1. package com.pattern.observer;
- 2.

```
3. public interface Observer {  
4.  
5. public abstract void update(NumberGenerator generator);  
6.  
7. }
```

2.NumberGenerator

[java] [view plain](#)[copy](#)[print?](#)

```
1. package com.pattern.observer;  
2.  
3. import java.util.ArrayList;  
4.  
5. import java.util.Iterator;  
6.  
7. /**  
8.  
9.  * @project JavaPattern  
10.  
11. * @author sunnylocus  
12.  
13. * @version 1.0.0  
14.  
15. * @date Aug 27, 2008 1:35:34 PM  
16.  
17. * @description 产生数值的抽象类  
18.  
19. */  
20.  
21. public abstract class NumberGenerator {  
22.  
23. private ArrayList observers = new ArrayList(); //存储 Observer  
24.  
25. /** 添加观察者*/
```

```
26.
27. public void addObserver(Observer observer) {
28.
29.     observers.add(observer);
30.
31. }
32.
33. /** 删除观察者*/
34.
35. public void delObserver(Observer observer) {
36.
37.     observers.remove(observer);
38.
39. }
40.
41. /** 通知所有观察者*/
42.
43. public void notifyObservers() {
44.
45.     Iterator it = observers.iterator();
46.
47.     while(it.hasNext()) {
48.
49.         Observer o =(Observer) it.next();
50.
51.         o.update(this); //this 相当于上面提到的邮局名
52.
53.     }
54.
55. }
56.
57. public abstract int getNumber(); //获取数字
58.
59. public abstract void generate(); //产生数字
60.
```

61.}

3.RandomNumberGenerator

[html] [view plaincopyprint?](#)

```
1. package com.pattern.observer;
2.
3. import java.util.Random;
4.
5. /**
6.
7.  * @project JavaPattern
8.
9.  * @author sunnylocus
10.
11.  * @version 1.0.0
12.
13.  * @date Aug 27, 2008 1:48:03 PM
14.
15.  * @description 用于产生随机数及通知观察者的类
16.
17. */
18.
19. public class RandomNumberGenerator extends NumberGenerator{
20.
21.     private Random random = new Random();//随机数产生器
22.
23.     private int number; //用于存放数字
24.
25.     public void generate() {
26.
27.         for(int i = 0; i < 5; i++) {
28.
29.             number = random.nextInt(10);//产生 10 以内的随机数
30.

```

```

31.notifyObservers(); //有新产生的数字，通知所有注册的观察者
32.
33.}
34.
35.}
36.
37.  /** 获得数字*/
38.
39.public int getNumber() {
40.
41.return number;
42.
43.}
44.
45.}

```

4.NumberObserver

[html] [view plaincopyprint?](#)

```

1. package com.pattern.observer;
2.
3. /** 以数字表示观察者的类*/
4.
5. public class NumberObserver implements Observer{
6.
7. public void update(NumberGenerator generator) {
8.
9. System.out.println("NumberObserver:"+ generator.getNumber());
10.
11.try {
12.
13.Thread.sleep(1000 * 3); //为了能清楚的看到输出，休眠 3 秒钟。
14.
15.}catch(InterruptedException e) {
16.

```



```

17.e.printStackTrace();
18.
19.}
20.
21.}
22.
23.}

```

5.SymbolObserver

[html] [view plain](#)[copy](#)[print?](#)

```

1. package com.pattern.observer;
2.
3. /** 以符号表示观察者的类*/
4.
5. public class SymbolObserver implements Observer{
6.
7.     public void update(NumberGenerator generator) {
8.
9.         System.out.print("SymbolObserver:");
10.
11.         int count = generator.getNumber();
12.
13.         for(int i = 0; i < count; i ++){
14.
15.             System.out.print("**^_^* ");
16.
17.         }
18.
19.         System.out.println("");
20.
21.         try {
22.
23.             Thread.sleep(1000 * 3);
24.

```

```
25.}catch(InterruptedExcepton e){
26.
27.e.printStackTrace();
28.
29.}
30.
31.}
32.
33.}
```

6.Main(测试类)

[html] [view plaincopyprint?](#)

```
1. package com.pattern.observer;
2.
3. public class Main {
4.
5.     public static void main(String[] args) {
6.
7.         //实例化数字产生对象
8.
9.         NumberGenerator generator =         RandomNumberGenerator();
10.
11.        //实例化观察者
12.
13.        Observer observer1 =         NumberObserver();
14.
15.        Observer observer2 =         SymbolObserver();
16.
17.        //注册观察者
18.
19.        generator.addObserver(observer1);
20.
21.        generator.addObserver(observer2);
```

```

22.
23.generator.generate(); //产生数字
24.
25.}
26.
27.}

```

7.结果输出

```

<terminated> Main (5) [Java Application] C:\jdk1.4.2_17\bin\javaw.exe (Aug 27, 2008 2:26:42 PM)
NumberObserver:4
SymbolObserver:*^_^*   *^_^*   *^_^*   *^_^*
NumberObserver:3
SymbolObserver:*^_^*   *^_^*   *^_^*
NumberObserver:8
SymbolObserver:*^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*
NumberObserver:2
SymbolObserver:*^_^*   *^_^*
NumberObserver:8
SymbolObserver:*^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*

```

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

Observer 模式是行为模式之一，它的作用是当一个对象的状态发生变化时，能够自动通知其他关联对象，自动刷新对象状态。Observer 模式提供给关联对象一种同步通信的手段，使某个对象与依赖它的其他对象之间保持状态同步。《设计模式》一书对 Observer 是这样描述的：**定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知并自动更新。**别名：依赖(Dependents)、发布-订阅(Publish-Subscribe)。

下面我们就来看看观察者模式的组成部分。

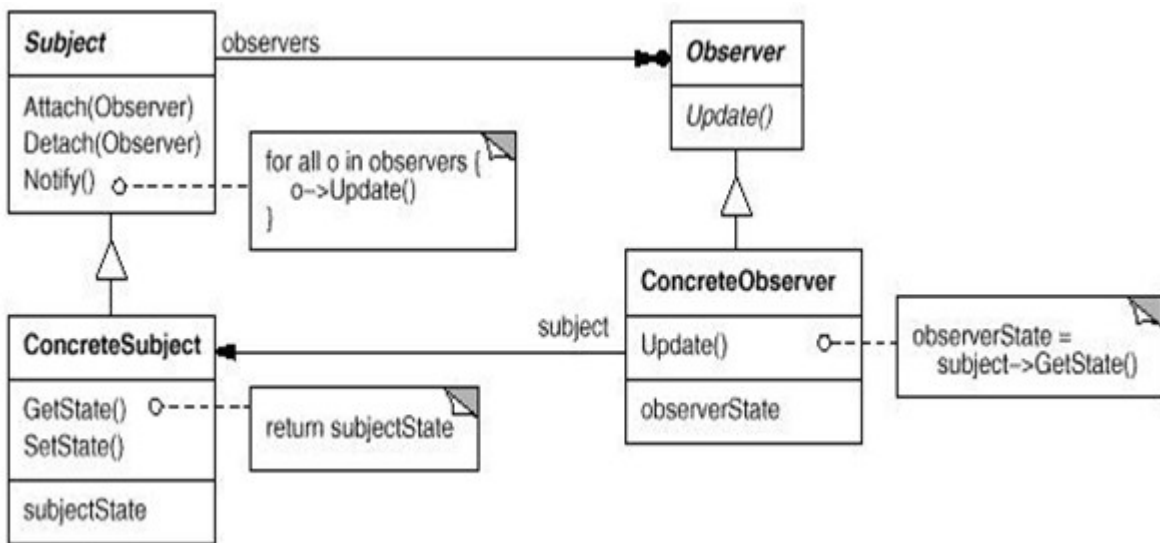
1) 抽象目标角色 (Subject)：目标角色知道它的观察者，可以有任意多个观察者观察同一个目标。并且提供注册和删除观察者对象的接口。目标角色往往由抽象类或者接口来实现。

2) 抽象观察者角色 (Observer)：为那些在目标发生改变时需要获得通知的对象定义一个更新接口。抽象观察者角色主要由抽象类或者接口来实现。

3) 具体目标角色 (Concrete Subject)：将有关状态存入各个 Concrete Observer 对象。当它的状态发生改变时，向它的各个观察者发出通知。

4) 具体观察者角色 (Concrete Observer)：存储有关状态，这些状态应与目标的状态保持一致。实现 Observer 的更新接口以使自身状态与目标的状态保持一致。在本角色内也可以维护一个指向 Concrete Subject 对象的引用。

结构如下：



观察者模式的优缺点：

观察者模式的效果有以下的优点：

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知，

观察者模式有下面的缺点：

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

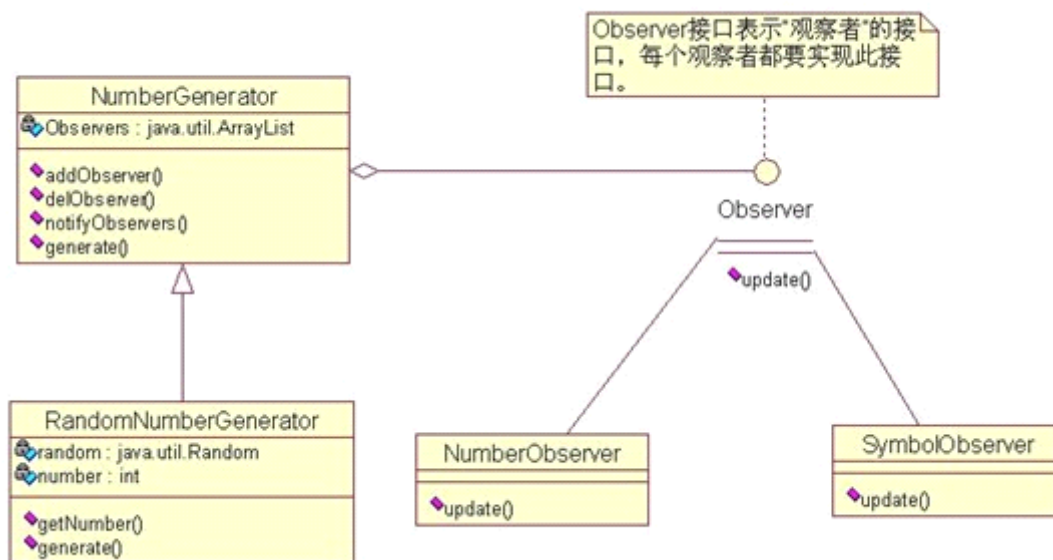
第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

下面用程序来说明一下观察者模式的流程：

一个随机数产生对象和两个观察者，这两个观察者都在随机数产生对象那里注册了，意思说如果你产生了新的数字，就通知我一声。

结构图：



类说明

名称

功能说明

Observer

表示观察者的接口，要成为观察者必须实现此接口才行

NumberGenerator

表示产生数值的抽象类

RandomNumberGenerator

产生随机数的类，继承于
NumberGenerator

NumberObserver

数字观察者，会打印出变化的数字

SymbolObserver

符号观察者,打印 *N* 个符号,打印多少个符号，由接受到的数值确定

1.Observer

[java] [view plaincopyprint?](#)

1. package com.pattern.observer;
- 2.

```
3. public interface Observer {  
4.  
5. public abstract void update(NumberGenerator generator);  
6.  
7. }
```

2.NumberGenerator

[java] [view plain](#)[copy](#)[print?](#)

```
1. package com.pattern.observer;  
2.  
3. import java.util.ArrayList;  
4.  
5. import java.util.Iterator;  
6.  
7. /**  
8.  
9.  * @project JavaPattern  
10.  
11. * @author sunnylocus  
12.  
13. * @version 1.0.0  
14.  
15. * @date Aug 27, 2008 1:35:34 PM  
16.  
17. * @description 产生数值的抽象类  
18.  
19. */  
20.  
21. public abstract class NumberGenerator {  
22.  
23. private ArrayList observers = new ArrayList(); //存储 Observer  
24.  
25. /** 添加观察者*/
```



```
26.
27. public void addObserver(Observer observer) {
28.
29.     observers.add(observer);
30.
31. }
32.
33. /** 删除观察者*/
34.
35. public void delObserver(Observer observer) {
36.
37.     observers.remove(observer);
38.
39. }
40.
41. /** 通知所有观察者*/
42.
43. public void notifyObservers() {
44.
45.     Iterator it = observers.iterator();
46.
47.     while(it.hasNext()) {
48.
49.         Observer o =(Observer) it.next();
50.
51.         o.update(this); //this 相当于上面提到的邮局名
52.
53.     }
54.
55. }
56.
57. public abstract int getNumber(); //获取数字
58.
59. public abstract void generate(); //产生数字
60.
```

61.}

3.RandomNumberGenerator

[html] [view plaincopyprint?](#)

```
1. package com.pattern.observer;
2.
3. import java.util.Random;
4.
5. /**
6.
7.  * @project JavaPattern
8.
9.  * @author sunnylocus
10.
11.  * @version 1.0.0
12.
13.  * @date Aug 27, 2008 1:48:03 PM
14.
15.  * @description 用于产生随机数及通知观察者的类
16.
17. */
18.
19. public class RandomNumberGenerator extends NumberGenerator{
20.
21.     private Random random = new Random();//随机数产生器
22.
23.     private int number; //用于存放数字
24.
25.     public void generate() {
26.
27.         for(int i = 0; i < 5; i++) {
28.
29.             number = random.nextInt(10);//产生 10 以内的随机数
30.

```

```

31.notifyObservers(); //有新产生的数字，通知所有注册的观察者
32.
33.}
34.
35.}
36.
37.  /** 获得数字*/
38.
39.public int getNumber() {
40.
41.return number;
42.
43.}
44.
45.}

```

4.NumberObserver

[html] [view plaincopyprint?](#)

```

1. package com.pattern.observer;
2.
3. /** 以数字表示观察者的类*/
4.
5. public class NumberObserver implements Observer{
6.
7. public void update(NumberGenerator generator) {
8.
9. System.out.println("NumberObserver:"+ generator.getNumber());
10.
11.try {
12.
13.Thread.sleep(1000 * 3); //为了能清楚的看到输出，休眠 3 秒钟。
14.
15.}catch(InterruptedException e) {
16.

```

```
17.e.printStackTrace();
18.
19.}
20.
21.}
22.
23.}
```

5.SymbolObserver

[html] [view plaincopyprint?](#)

```
1. package com.pattern.observer;
2.
3. /** 以符号表示观察者的类*/
4.
5. public class SymbolObserver implements Observer{
6.
7.     public void update(NumberGenerator generator) {
8.
9.         System.out.print("SymbolObserver:");
10.
11.         int count = generator.getNumber();
12.
13.         for(int i = 0; i < count; i++) {
14.
15.             System.out.print("**^_^* ");
16.
17.         }
18.
19.         System.out.println("");
20.
21.         try {
22.
23.             Thread.sleep(1000 * 3);
```

```
24.  
25.}catch(InterruptedExceptio e){  
26.  
27.e.printStackTrace();  
28.  
29.}  
30.  
31.}  
32.  
33.}
```

6.Main(测试类)

[\[html\] view plaincopyprint?](#)

```
1. package com.pattern.observer;  
2.  
3. public class Main {  
4.  
5. public static void main(String[] args) {  
6.  
7. //实例化数字产生对象  
8.  
9. NumberGenerator generator =      RandomNumberGenerator();  
10.  
11.//实例化观察者  
12.  
13.Observer observer1 =      NumberObserver();  
14.  
15.Observer observer2 =      SymbolObserver();  
16.  
17.//注册观察者  
18.  
19.generator.addObserver(observer1);  
20.
```

```

21.generator.addObserver(observer2);
22.
23.generator.generate(); //产生数字
24.
25.}
26.
27.}

```

7.结果输出

```

<terminated> Main (5) [Java Application] C:\jdk1.4.2_17\bin\javaw.exe (Aug 27, 2008 2:26:42 PM)
NumberObserver:4
SymbolObserver:*^_^*   *^_^*   *^_^*   *^_^*
NumberObserver:3
SymbolObserver:*^_^*   *^_^*   *^_^*
NumberObserver:8
SymbolObserver:*^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*
NumberObserver:2
SymbolObserver:*^_^*   *^_^*
NumberObserver:8
SymbolObserver:*^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*   *^_^*

```

（三十七）细谈 struts2（二）开发第一个 struts2 的实例

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

前面一篇博客（[细谈 struts2 之自己实现 struts2 框架](#)）带大家对于基于 mvc 业务流程熟悉了一下，现在我们就用对 mvc 实现最好的框架 struts2 来开发一个应用实例。虽然现在 MyEclipse8.5 以上版本已经开始支持 Struts2，但为了我们能更好的熟悉开发 struts2 的业务流程，现在我们还是手动去搭配环境。首先我们需要到 struts.apache.org 去下载 **struts-2.2.3-all** 包。现在最高版本应该达到 2.3 了。要想正常使用 Struts2，至少需要如下五个包（可能会因为 Struts2 的版本不同，包名略有差异，但包名的前半部是一样的）。

struts2-core-2.0.11.1.jar

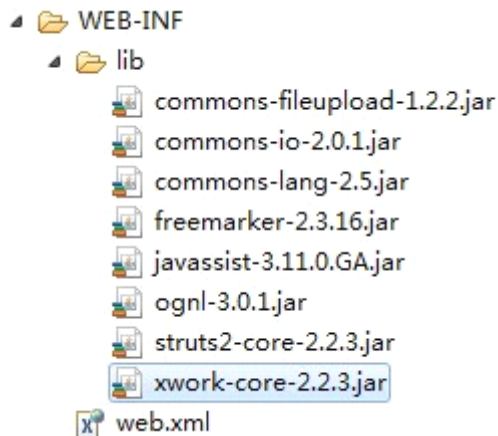
xwork-2.0.4.jar

commons-logging-1.0.4.jar

freemarker-2.3.8.jar

ognl-2.6.11.jar

注：貌似好像一些高版本的还需要加入一些其他 jar 包，如下图所示：



好了，jar 包加入之后，我们下一步开始搭配配置环境了。很多同学可能会有这样的疑问，为什么我提交的请求能到 struts.xml 去找对应的 action 呢？？至少我刚开始学习的时候有这么个疑问。现在答案即可以为大家揭晓了，因为 struts2 的核心是拦截器，一切请求都要经过拦截器才转发给所对应的 action 的。Struts2 中第一个拦截请求的就是

org.apache.struts2.dispatcher.FilterDispatcher 这个拦截器（**下一篇博客我们将对这个拦截器的源码进行分析**），拦截器对请求进行一些处理之后然后去 **struts.xml** 寻找对应的 **action**。我们一起来看一下 **web.xml** 的配置：

[html] view plaincopyprint?

1. `<?xml version= encoding= ?>`
2. `<web-app version=`
3. `xmlns=`
4. `xmlns:xsi=`
5. `xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee`
6. `http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">`
7. `<filter>`
8. `<filter-name>struts2</filter-name>`
9. `<filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>`
10. `</filter>`


```

11. <filter-mapping>
12. <filter-name>struts2</filter-name>
13. <url-pattern>*.action</url-pattern>
14. </filter-mapping>
15. <welcome-file-list>
16. <welcome-file>index.jsp</welcome-file>
17. </welcome-file-list>
18. </web-app>

```

在 struts2 官方提供的文档中要求,在服务器 class 文件夹下建立 struts.xml 文件。由于在 web 项目部署到服务器上,开启服务器对 web 项目进行编译时会自动把 src 文件夹下的文件加载到服务器 class 文件夹下,所以我们直接在 src 下面建立 struts.xml 文件, **具体 struts.xml 配置**如下:

[html] [view plain](#)[copy](#)[print?](#)

```

1. <?xml version=      encoding=      ?>
2. <!DOCTYPE struts PUBLIC
3.     "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
4.     "http://struts.apache.org/dtds/struts-2.0.dtd">
5. <struts>
6.   <constant name=      value=      />
7.   <package name=      extends=      >
8.     <action name=      class=      >
9.       <result name=      ></result>
10.      <result name=      ></result>
11.    </action>
12.  </package>
13. </struts>

```

注：上述代码具体意义：

1.<constant>标签主要是用来修改 struts.properties 配置文件信息, name 和 value 分别相当于 struts.properties 文件中的 name 和 value

2.<package>主要是作为分包作用，比如一个项目分好几个模块，这里可以每一个模块分配一个包，一个 struts.xml 文件中可以出现多个<package>标签，这里一定要有 extends="struts-default"，因为 struts 的核心拦截器都配置在 struts-default 包中，如果没有这个，所有请求都不会请求到

3.一个<action>标签对应一个 action 类，主要是通过 action 标签中的去寻找 class,然后执行对应的 class.Action 标签里有一个一个 method 属性，他可以指定执行 action 中的哪个方法

下面我们就开始以登录为例来写一下 struts2 开发的视图层：

Login.jsp

[html] view plaincopyprint?

```
1. <%@ page language=      import=      pageEncoding=      %>
2. <%
3. String path =      .getContextPath();
4. String basePath =      .getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";
5. %>
6. <html>
7. <head>
8. </head>
9. <body>
10. <form action=      >
11. <input type=      name=      /><br/>
12. <input type=      name=      /><br/>
13. <input type=      value=      />
14. </form>
15. </body>
16. </html>
```

好了，视图层写完了，下面就要开始写核心 action 了，由于业务逻辑层就是判断用户名和密码是否与固定的 admin 和 123456 相等，所以本程序只是为了测试，就不再单独抽出来了，直接写在 action 里面了 LoginAction.java

[java] [view plaincopyprint?](#)

```
1. package com.bzu.action;
2. public class LoginAction {
3.
4.     private String username;
5.     private String password;
6.     public String getUsername() {
7.         return username;
8.     }
9.     public void setUsername(String username) {
10.        this.username = username;
11.    }
12.    public String getPassword() {
13.        return password;
14.    }
15.    public void setPassword(String password) {
16.        this.password = password;
17.    }
18.    public String execute(){
19.
20.        if(username.equals("admin")&&password.equals("123456"))
21.            return "success";
22.        return "fail";
23.    }
24.}
```

从上面的程序可以看出，我们在 action 中要把 form 表单中数据都以私有变量的形式定义出来，然后在提供对应的 set 和 get 方法。很多同学可能在这又有

疑问了。为什么给他提供 **set** 和 **get** 方法，**form** 表单中的数据就可以设置到对应的属性上呢？为什么他会默认的去执行 **execute** 方法呢？为什么把配置文件中 **action** 标签对应的 **method** 属性修改后就可以执行新设置的方法呢？

呵呵，在这在卖个关子，在接下来的博客中，会为大家一一把这些疑问解决。

Action 写完之后，我们就可以把 `struts.xml` 对应的写上了，本程序完整的

`struts.xml`:

[html] [view plaincopyprint?](#)

```
1. <?xml version=      encoding=      ?>
2. <!DOCTYPE struts PUBLIC
3.     "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
4.     "http://struts.apache.org/dtds/struts-2.0.dtd">
5. <struts>
6.   <constant name=      value=      />
7.   <package name=      extends=      >
8.     <action name=      class=      >
9.       <result name=      >success.jsp</result>
10.      <result name=      >fail.jsp</result>
11.    </action>
12.  </package>
13.</struts>
14.
```

对应的 `success.jsp` 和 `fai.jsp` 没什么内容，就是显示成功和失败几个字。

好了，到此，我们的第一个 `struts2` 的应用程序就写完了，下面我们一起来看一下运行结果：

http://localhost:8080/Struts2/login.jsp

username:
password:

—————>>

http://localhost:8080/Struts2/LoginAction.action?username=admin&password=123456

成功

（三十八）细谈 struts2（三）struts2 拦截器源码分析

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

前面博客我们介绍了开发 struts2 应用程序的基本流程（细谈 struts2 之开发第一个 struts2 的实例），通过前面我们知道了 struts2 实现请求转发和配置文件加载都是拦截器进行的操作，这也就是为什么我们要在 web.xml 配置 struts2 的拦截器的原因了。我们知道，在开发 struts2 应用开发的时候我们要在 web.xml 进行配置拦截器

org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter（在一些老版的一般配置 org.apache.struts2.dispatcher.FilterDispatcher），不知道大家刚开始学的时候有没有这个疑问，为什么通过这个拦截器我们就可以拦截到我们提交的请求，并且一些配置文件就可以得到加载呢？不管你有没有，反正我是有。我想这个问题的答案，我们是非常有必要去看一下这个拦截器的源码去找。

打开 StrutsPrepareAndExecuteFilter 拦截器源码我们可以看出以下类的信息
属性摘要：

Protected	List<Pattern>	excludedPatterns
protected	ExecuteOperations	execute
protected	PrepareOperations	prepare

我们可以看出 StrutsPrepareAndExecuteFilter 与普通的 Filter 并无区别，方法除继承自 Filter 外，仅有一个回调方法，第三部分我们将按照 Filter 方法调用顺序，init—>doFilter—>destroy 顺序地分析源码。

提供的方法：

destroy()

继承自 **Filter**，用于

资源释放

doFilter(ServletRequest req, ServletResponse res, FilterChain chain)

继承自 **Filter**，执行

方法

init(FilterConfig filterConfig)

继承自 **Filter**，初始化

参数

postInit(Dispatcher dispatcher, FilterConfig filterConfig)

Callback for post initialization（一个空的

方法，用于方法回调

初始化）

下面我们一一对这些方法看一下：

1.init 方法：我们先整体看一下这个方法：

[java] [view plaincopyprint?](#)

```
1. public void init(FilterConfig filterConfig) throws ServletException {
2.
3.     InitOperations init = new InitOperations();
4.
5.     try {
6.
7.         //封装 filterConfig，其中有个主要方法 getInitParameterNames 将参数名字以 String
           格式存储在 List 中
8.
9.         FilterHostConfig config = new FilterHostConfig(filterConfig);
10.
11. // 初始化 struts 内部日志
```

```

12.
13.init.initLogging(config);
14.
15.//创建 dispatcher ， 并初始化，这部分下面我们重点分析，初始化时加载那些资源
16.
17.Dispatcher dispatcher = init.initDispatcher(config);
18.
19.init.initStaticContentLoader(config, dispatcher);
20.
21.//初始化类属性： prepare 、 execute
22.
23.prepare = new PrepareOperations(filterConfig.getServletContext(), dispatcher);
24.
25.execute = new ExecuteOperations(filterConfig.getServletContext(), dispatcher);
26.
27.this.excludedPatterns = init.buildExcludedPatternsList(dispatcher);
28.
29.//回调空的 postInit 方法
30.
31.postInit(dispatcher, filterConfig);
32.
33.} finally {
34.
35.init.cleanup();
36.
37.}
38.
39.}

```

首先开一下 **FilterHostConfig** 这个封装 **configfilter** 的类： 这个类总共不超过二三十行代码 **getInitParameterNames** 是这个类的核心，将 **Filter** 初始化参数名称有枚举类型转为 **Iterator**。此类的主要作为是对 **filterConfig** 封装。具体代码如下：

[java] [view plaincopyprint?](#)

```
1. public Iterator<String> getInitParameterNames() {
2.
3.     return MakelIterator.convert(config.getInitParameterNames());
4.
5. }
```

下面咱接着一块看 `Dispatcher dispatcher = init.initDispatcher(config);`这是重点，创建并初始化 `Dispatcher` ，看一下具体代码：

[html] [view plaincopyprint?](#)

```
1. public Dispatcher initDispatcher( HostConfig filterConfig ) {
2.
3.     Dispatcher dispatcher =
4.
5.     dispatcher.init();
6.
7.     return dispatcher;
8.
9. }
10.
11.
```

```
12.SPAN style=
> </SPAN><SPAN style=
><SPAN style=
><STRONG>创建
<SPAN style=
>Dispatcher</SPAN><SPAN style=
>, 会读
取 </SPAN><SPAN style=
>filterConfig </SPAN><SPA
N style=
>中的配置信息，将配置信息解析出来，封装成为一
个
</SPAN><SPAN style=
>Map</SPAN></STRONG></
SPAN><SPAN style=
>, 然后
```

```

</SPAN></SPAN><SPAN style=
    >根据
</SPAN><SPAN style=
    >上下文和参数
servlet<SPAN style=
    >Map</SPAN><SPAN style=
    >构造
</SPAN><SPAN style=
    >Dispatcher </SPAN><SPAN
style=
    >: </SPAN></SPAN></SPAN>

```

[java] [view plain](#)copyprint?

```

1. private Dispatcher createDispatcher( HostConfig filterConfig ) {
2.
3. Map<String, String> params = new HashMap<String, String>();
4.
5. for ( Iterator e = filterConfig.getInitParameterNames(); e.hasNext(); ) {
6.
7. String name = (String) e.next();
8.
9. String value = filterConfig.getInitParameter(name);
10.
11.params.put(name, value);
12.
13.}
14.
15.return new Dispatcher(filterConfig.getServletContext(), params);
16.
17.}

```

Dispatcher 构造玩以后，开始对他进行初始化，加载 struts2 的相关配置文件，**将按照顺序逐一加载：default.properties, struts-default.xml, struts-plugin.xml, struts.xml,**我们一起看看他是怎么一步步的加载这些文件的 dispatcher 的 init()方法：

[html] [view plain](#)[copy](#)[print](#)?

```
1. public void init() {
2.
3.
4.     if (configurationManager == null) {
5.
6.         configurationManager =                      (BeanSelectionProvider.D
           EFAULT_BEAN_NAME);
7.
8.     }
9.     try {
10.
11.         init_DefaultProperties(); // [1]
12.
13.         init_TraditionalXmlConfigurations(); // [2]
14.
15.         init_LegacyStrutsProperties(); // [3]
16.
17.         init_CustomConfigurationProviders(); // [5]
18.
19.         init_FilterInitParameters() ; // [6]
20.
21.         init_AliasStandardObjects() ; // [7]
22.
23.
24.         Container container =                      ();
25.
26.         container.inject(this);
27.
28.         init_CheckConfigurationReloading(container);
29.
30.         init_CheckWebLogicWorkaround(container);
31.
32.
```

```

33.         if (!dispatcherListeners.isEmpty()) {
34.
35.             for (DispatcherListener l : dispatcherListeners) {
36.
37.                 l.dispatcherInitialized(this);
38.
39.             }
40.
41.         }
42.
43.     } catch (Exception ex) {
44.
45.         if (LOG.isErrorEnabled())
46.
47.             LOG.error("Dispatcher initialization failed", ex);
48.
49.         throw new StrutsException(ex);
50.
51.     }
52.
53. }

```

下面我们一起来看一下【1】，【2】，【3】，【5】，【6】的源码，看一下什么都一目了然了：

1.这个方法中是将一个 DefaultPropertiesProvider 对象追加到 ConfigurationManager 对象内部的 ConfigurationProvider 队列中。 DefaultPropertiesProvider 的 register()方法可以载入 org/apache/struts2/default.properties 中定义的属性。

[html] [view plaincopyprint?](#)

```

1. try {

```

```

2.
3.     defaultSettings = PropertiesSettings("org/apache/struts2/default");
4.
5.     } catch (Exception e) {
6.
7.         throw new ConfigurationException("Could not find or error in org/apache/struts2/default.properties", e);
8.
9.     }

```

2. 调用 `init_TraditionalXmlConfigurations()` 方法，实现载入 `FilterDispatcher` 的配置中所定义的 `config` 属性。如果用户没有定义 `config` 属性，`struts` 默认会载入 `DEFAULT_CONFIGURATION_PATHS` 这个值所代表的 xml 文件。它的值为 `"struts-default.xml, struts-plugin.xml, struts.xml"`。也就是说框架默认会载入这三个项目 xml 文件。如果文件类型是 XML 格式，则按照 `xwork-x.x.dtd` 模板进行读取。如果，是 Struts 的配置文件，则按 `struts-2.X.dtd` 模板进行读取。

```
private static final String DEFAULT_CONFIGURATION_PATHS = "struts-
default.xml, struts-plugin.xml, struts.xml";
```

3. 创建一个 `LegacyPropertiesConfigurationProvider` 类，并将它追加到 `ConfigurationManager` 对象内部的 `ConfigurationProvider` 队列中。

`LegacyPropertiesConfigurationProvider` 类载入 `struts.properties` 中的配置，这个文件中的配置可以覆盖 `default.properties` 中的。其子类是

`DefaultPropertiesProvider` 类

5. `init_CustomConfigurationProviders()` 此方法处理的是 `FilterDispatcher` 的配置中所定义的 `configProviders` 属性。负责载入用户自定义的 `ConfigurationProvider`。

[html] [view plain](#)copyprint?

```
1. String configProvs =          .get("configProviders");
2.
3.     if (configProvs != null) {
4.
5.         String[] classes =          .split("\\s*[,]\\s*");
6.
7.         for (String cname : classes) {
8.
9.             try {
10.
11.                 Class cls =          .loadClass(cname, this.getClass());
12.
13.                 ConfigurationProvider prov = (ConfigurationProvider)cls.newInstance()
14.                 ;
15.                 configurationManager.addConfigurationProvider(prov);
16.
17.             } catch (InstantiationException e) {
18.
19.                 throw new ConfigurationException("Unable to instantiate provider: "+c
20.                 name, e);
21.
22.             } catch (IllegalAccessException e) {
23.
24.                 throw new ConfigurationException("Unable to access provider: "+cna
25.                 me, e);
26.
27.             } catch (ClassNotFoundException e) {
28.
29.                 throw new ConfigurationException("Unable to locate provider class: "
30.                 +cname, e);
31.             }
```

```
31.    }  
32.  
33.    }
```

6.`init_FilterInitParameters()`此方法用来处理 `FilterDispatcher` 的配置中所定义的所有属性

7. `init_AliasStandardObjects()`，将一个 `BeanSelectionProvider` 类追加到 `ConfigurationManager` 对象内部的 `ConfigurationProvider` 队列中。

`BeanSelectionProvider` 类主要实现加载 `org/apache/struts2/struts-messages`。

[\[html\] view plaincopyprint?](#)

```
1. private void init_AliasStandardObjects() {  
2.     configurationManager.addConfigurationProvider(  
3.     new BeanSelectionProvider());  
4. }
```

相信看到这大家应该明白了，**struts2** 的一些配置的加载顺序和加载时所做的工作，其实有些地方我也不是理解的很清楚。其他具体的就不在说了，`init` 方法暂时先介绍到这

2、`doFilter` 方法

`doFilter` 是过滤器的执行方法，它拦截提交的 `HttpServletRequest` 请求，`HttpServletResponse` 响应，作为 **struts2** 的核心拦截器，在 `doFilter` 里面到底做了哪些工作，我们将逐行解读其源码，大体源码如下：

[\[html\] view plaincopyprint?](#)

```

1. public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {
2.
3. //父类向子类转：强转为 http 请求、响应
4.
5. HttpServletRequest request = (HttpServletRequest) req;
6.
7. HttpServletResponse response = (HttpServletResponse) res;
8.
9. try {
10.
11. //设置编码和国际化
12.
13. prepare.setEncodingAndLocale(request, response);
14.
15. //创建 Action 上下文（重点）
16.
17. prepare.createActionContext(request, response);
18.
19. prepare.assignDispatcherToThread();
20.
21. if ( excludedPatterns != null && prepare.isUrlExcluded(request, excludedPatterns))
22. {
23. chain.doFilter(request, response);
24.
25. } else {
26.
27. request =          .wrapRequest(request);
28.
29. ActionMapping mapping =          .findActionMapping(request, response, true);
30.
31. if (mapping == null) {
32.
33. boolean handled =          .executeStaticResourceRequest(request, response);

```



```
34.  
35.if (!handled) {  
36.  
37.chain.doFilter(request, response);  
38.  
39.}  
40.  
41.} else {  
42.  
43.execute.executeAction(request, response, mapping);  
44.  
45.}  
46.  
47.}  
48.  
49.} finally {  
50.  
51.prepare.cleanupRequest(request);  
52.  
53.}  
54.  
55.}
```

下面我们就逐句的来的看一下：设置字符编码和国际化很简单 `prepare` 调用了 `setEncodingAndLocale`，然后调用了 `dispatcher` 方法的 `prepare` 方法：

[html] [view plaincopyprint?](#)

```
1. /**  
2.  
3. * Sets the request encoding and locale on the response  
4.  
5. */  
6.
```

```

7. public void setEncodingAndLocale(HttpServletRequest request, HttpServletResponse response) {
8.
9.     dispatcher.prepare(request, response);
10.
11.}

```

看下 **prepare** 方法，这个方法很简单只是设置了 **encoding** 、 **locale** ，做的只是一些辅助的工作：

[html] [view plaincopyprint?](#)

```

1. public void prepare(HttpServletRequest request, HttpServletResponse response) {
2.
3.     String encoding =    ;
4.
5.     if (defaultEncoding != null) {
6.
7.         encoding =    ;
8.
9.     }
10.
11.     Locale locale =    ;
12.
13.     if (defaultLocale != null) {
14.
15.         locale =    .getLocaleFrom(defaultLocale, request.getLocale());
16.
17.     }
18.
19.     if (encoding != null) {
20.
21.     try {
22.

```

```
23.request.setCharacterEncoding(encoding);
24.
25.} catch (Exception e) {
26.
27.LOG.error("Error setting character encoding to '" + encoding + "' - ignoring.", e);
28.
29.}
30.
31.}
32.
33.if (locale != null) {
34.
35.response.setLocale(locale);
36.
37.}
38.
39.if (paramsWorkaroundEnabled) {
40.
41.request.getParameter("foo"); // simply read any parameter (existing or not) to "prime" the request
42.
43.}
44.
45.}
```

下面咱重点看一下创建 Action 上下文重点

Action 上下文创建（重点）

ActionContext 是一个容器，这个容易主要存储 request、session、application、parameters 等相关信息.ActionContext 是一个线程的本地变量，这意味着不同的 action 之间不会共享 ActionContext，所以也不用

考虑线程安全问题。其实质是一个 Map, key 是标示 request、session、..... 的字符串, 值是其对应的对象:

```
static ThreadLocal actionContext = new ThreadLocal();
Map<String, Object> context;
```

下面我们看起来下创建 action 上下文的源码:

[\[html\] view plaincopyprint?](#)

```
1. /**
2.
3.  *创建 Action 上下文, 初始化 thread local
4.
5. */
6.
7. public ActionContext createActionContext(HttpServletRequest request, HttpServle
    tResponse response) {
8.
9.     ActionContext ctx;
10.
11.     Integer counter = ;
12.
13.     Integer oldCounter = (Integer) request.getAttribute(CLEANUP_RECURSION_CO
        UNTER);
14.
15.     if (oldCounter != null) {
16.
17.         counter = + 1;
18.
19.     }
20.
21.
22.
23. //注意此处是从 ThreadLocal 中获取此 ActionContext 变量
24.
25. ActionContext oldContext = .getContext();
```

```

26.
27.if (oldContext != null) {
28.
29.// detected existing context, so we are probably in a forward
30.
31.ctx =      ActionContext(new HashMap<String, Object>(oldContext.getContextM
    ap()));
32.
33.} else {
34.
35.ValueStack stack =      .getContainer().getInstance(ValueStackFactory.clas
    s).createValueStack();
36.
37.stack.getContext().putAll(dispatcher.createContextMap(request, response, null, se
    rvletContext));
38.
39.//stack.getContext()返回的是一个 Map<String, Object>, 根据此 Map 构造一个
    ActionContext
40.
41.ctx =      ActionContext(stack.getContext());
42.
43.}
44.
45.request.setAttribute(CLEANUP_RECURSION_COUNTER, counter);
46.
47.//将 ActionContext 存如 ThreadLocal
48.
49.ActionContext.setContext(ctx);
50.
51.return ctx;
52.
53.}

```

一句句来看：

```
ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.class).createValueStack();
```

`dispatcher.getContainer().getInstance(ValueStackFactory.class)` 根据字面估计一下就是创建 `ValueStackFactory` 的实例。这个地方我也只是根据字面来理解的。`ValueStackFactory` 是接口，其默认实现是 `OgnlValueStackFactory`，调用 `OgnlValueStackFactory` 的 `createValueStack()`：

下面看一下 `OgnlValueStack` 的构造方法

[\[html\] view plaincopyprint?](#)

```
1. protected OgnlValueStack(XWorkConverter xworkConverter, CompoundRootAccessor accessor, TextProvider prov, boolean allowStaticAccess) {
2.
3.     //new 一个 CompoundRoot 出来
4.
5.     setRoot(xworkConverter, accessor, new CompoundRoot(), allowStaticAccess);
6.
7.     push(prov);
8.
9. }
```

接下来看一下 `setRoot` 方法：

[\[html\] view plaincopyprint?](#)

```
1. protected void setRoot(XWorkConverter xworkConverter, CompoundRootAccessor accessor, CompoundRoot compoundRoot,
2.
3.     boolean allowStaticMethodAccess) {
4.
5.     //OgnlValueStack.root =
6.
7.     this.root =
```

```

8.
9. 1 //方法/属性访问策略。
10.
11. this.securityMemberAccess = SecurityMemberAccess(allowStaticMethodAccess);
12.
13. //创建 context 了，创建 context 使用的是 ognl 的默认方式。
14.
15. //Ognl.createDefaultContext 返回一个 OgnlContext 类型的实例
16.
17. //这个 OgnlContext 里面，root 是 OgnlValueStack 中的 compoundRoot，map 是
    OgnlContext 自己创建的 private Map _values = HashMap(23);
18.
19. this.context = .createDefaultContext(this.root, accessor, new OgnlTypeConverterWrapper(xworkConverter), securityMemberAccess);
20.
21.
22.
23. //不是太理解，猜测如下：
24.
25. //context 是刚刚创建的 OgnlContext，其中的 HashMap 类型 _values 加入如下
    k-v:
26.
27. //key:com.opensymphony.xwork2.util.ValueStack.ValueStack
28.
29. //value:this,这个应该是当前的 OgnlValueStack 实例。
30.
31. //刚刚用断点跟了一下，_values 里面是：
32.
33. //com.opensymphony.xwork2.ActionContext.container= .opensymphony.xwork2.inject.ContainerImpl@96231e
34.
35. //com.opensymphony.xwork2.util.ValueStack.ValueStack= .opensymphony.xwork2.ognl.OgnlValueStack@4d912
36.

```

```

37. context.put(VALUE_STACK, this);
38.
39. //此时: OgnlValueStack 中的 compoundRoot 是空的;
40.
41. //context 是一个 OgnlContext, 其中的_root 指向 OgnlValueStack 中的 root,
    _values 里面的东西, 如刚才所述。
42.
43. //OgnlContext 中的额外设置。
44.
45. Ognl.setClassResolver(context, accessor);
46.
47. ((OgnlContext) context).setTraceEvaluations(false);
48.
49. ((OgnlContext) context).setKeepLastEvaluation(false);
50.
51.}
52.

```

上面代码中 `dispatcher.createContextMap`, 如何封装相关参数: , 我们以 `RequestMap` 为例, 其他的原理都一样: 主要方法实现:

[html] [view plaincopyprint?](#)

```

1. //map 的 get 实现
2.
3. public Object get(Object key) {
4.
5. return request.getAttribute(key.toString());
6.
7. }
8.
9. //map 的 put 实现
10.
11. public Object put(Object key, Object value) {
12.
13. Object oldValue = (key);

```



```
14.  
15. entries =    ;  
16.  
17. request.setAttribute(key.toString(), value);  
18.  
19. return oldValue;  
20.  
21.}
```

到此，几乎 **StrutsPrepareAndExecuteFilter** 大部分的源码都涉及到了。
自己感觉都好乱，所以还请大家见谅，能力有限，希望大家可以共同学习

（三十九）大话设计模式（七）代理模式和 java 动态代理机制

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

代理设计模式

代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。代理类负责为委托类预处理消息，过滤消息并转发消息，以及进行消息被委托类执行后的后续处理。

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

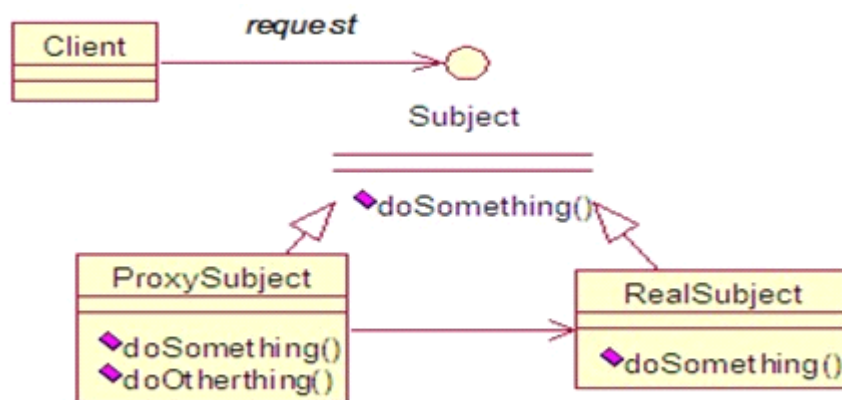
代理模式一般涉及到的角色有：

抽象角色：声明真实对象和代理对象的共同接口；

代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象

图 1. 代理模式类图



为了保持行为的一致性，代理类和委托类通常会实现相同的接口，所以在访问者看来两者没有丝毫的区别。通过代理类这中间一层，能有效控制对委托类对象的直接访问，也可以很好地隐藏和保护委托类对象，同时也为实施不同控制策略预留了空间，从而在设计上获得了更大的灵活性。**Java** 动态代理机制以巧妙的方式近乎完美地实践了代理模式的设计理念。

java 动态代理

相关的类和接口

要了解 **Java** 动态代理的机制，首先需要了解以下相关的类或接口：

- **java.lang.reflect.Proxy**：这是 **Java** 动态代理机制的主类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。

清单 1. Proxy 的静态方法

[\[html\] view plaincopyprint?](#)

1. // 方法 1: 该方法用于获取指定代理对象所关联的调用处理器
- 2.
3. static InvocationHandler getInvocationHandler(Object proxy)
- 4.
5. // 方法 2: 该方法用于获取关联于指定类装载器和一组接口的动态代理类的类对象
- 6.
7. static Class getProxyClass(ClassLoader loader, Class[] interfaces)

- 8.
9. // 方法 3: 该方法用于判断指定类对象是否是一个动态代理类
- 10.
11. static boolean isProxyClass(Class cl)
- 12.
13. // 方法 4: 该方法用于为指定类装载机、一组接口及调用处理器生成动态代理类实例
- 14.
15. static Object newProxyInstance(ClassLoader loader, Class[] interfaces,
- 16.
17. InvocationHandler h)

`java.lang.reflect.InvocationHandler`: 这是调用处理器接口，它自定义了一个 `invoke` 方法，用于集中处理在动态代理类对象上的方法调用，通常在该方法中实现对委托类的代理访问。

清单 2. `InvocationHandler` 的核心方法

[\[html\] view plaincopyprint?](#)

1. // 该方法负责集中处理动态代理类上的所有方法调用。第一个参数既是代理类实例，第二个参数是被调用的方法对象
- 2.
3. // 第三个方法是调用参数。调用处理器根据这三个参数进行预处理或分派到委托类实例上发射执行
- 4.
5. Object invoke(Object proxy, Method method, Object[] args)

每次生成动态代理类对象时都需要指定一个实现了该接口的调用处理器对象（参见 `Proxy` 静态方法 4 的第三个参数）。

· `java.lang.ClassLoader`: 这是类装载机类，负责将类的字节码装载

到 Java 虚拟机（JVM）中并为其定义类对象，然后该类才能被使用。

`Proxy` 静态方法生成动态代理类同样需要通过类装载机来进行装载才能使

用,它与普通类的唯一区别就是其字节码是由 JVM 在运行时动态生成的而非预存在于任何个 .class 文件中。

每次生成动态代理类对象时都需要指定一个类装载器对象（参见 Proxy 静态方法 4 的第一个参数）

代理机制及其特点

首先让我们来了解一下如何使用 Java 动态代理。具体有如下四步骤：

1. 通过实现 InvocationHandler 接口创建自己的调用处理器；
2. 通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理类；
3. 通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型；
4. 通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。

清单 3. 动态代理对象创建过程

[html] [view plaincopyprint?](#)

1. // InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托类的分派转发
- 2.
3. // 其内部通常包含指向委托类实例的引用，用于真正执行分派转发过来的方法调用
- 4.
5. InvocationHandler handler = InvocationHandlerImpl(..);
- 6.
- 7.
8. // 通过 Proxy 为包括 Interface 接口在内的一组接口动态创建代理类的类对象
- 9.

```

10. Class clazz =      .getProxyClass(classLoader, new Class[] { Interface.class, ... })
    ;
11.
12.
13. // 通过反射从生成的类对象获得构造函数对象
14.
15. Constructor constructor =      .getConstructor(new Class[] { InvocationHandler.cl
    ass });
16.
17.
18. // 通过构造函数对象创建动态代理类实例
19.
20. Interface Proxy = (Interface)constructor.newInstance(new Object[] { handler });

```

实际使用过程更加简单，因为 Proxy 的静态方法 newProxyInstance 已经为我们封装了步骤 2 到步骤 4 的过程，所以简化后的过程如

清单 4. 简化的动态代理对象创建过程

[html] [view plaincopyprint?](#)

```

1. // InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理
    类到委托类的分派转发
2.
3. InvocationHandler handler =      InvocationHandlerImpl(..);
4.
5. // 通过 Proxy 直接创建动态代理类实例
6.
7. Interface proxy = (Interface)Proxy.newProxyInstance( classLoader,
8.
9. new Class[] { Interface.class },
10.
11. handler );

```

下面我们来看一个简单实现动态代理的例子：

1.代理类和真实类接口：

[html] [view plaincopyprint?](#)

```
1. public interface Subject
2.
3. {
4.
5.     public void request();
6.
7. }
```

2.真实类：

[html] [view plaincopyprint?](#)

```
1. public class RealSubject implements Subject
2.
3. {
4.
5.     public void request()
6.
7. {
8.
9.     System.out.println("From real subject!");
10.
11.}}
```

3.具体代理类：

[html] [view plaincopyprint?](#)

```
1. import java.lang.reflect.InvocationHandler;
2.
3. import java.lang.reflect.Method;
4.
```

```
5. public class DynamicSubject implements InvocationHandler
6.
7. {
8.
9.     private Object sub;
10.
11. public DynamicSubject(Object obj)
12.
13. {
14.
15.     this.sub = obj;
16.
17. }
18.
19. public Object invoke(Object proxy, Method method, Object[] args)
20.
21. throws Throwable
22.
23. {
24.
25.     System.out.println("before calling: " + method);
26.
27.     method.invoke(sub, args);
28.
29.     System.out.println("after calling: " + method);
30.
31.     return method.invoke(sub, args);
32.
33. }
34.
35. }
```

注：该代理类的内部属性是 **Object** 类型，实际使用的时候通过该类的构造方法传递进来一个对象。此外，该类还实现了 **invoke** 方法，该方法中的

method.invoke 其实就是调用被代理对象的将要执行的方法，方法参数是 **sub**，表示该方法从属于 **sub**，通过动态代理类，我们可以在执行真实对象的方法前后加入自己的一些额外方法。

4.客户端调用示例：

[html] [view plaincopyprint?](#)

```
1. import java.lang.reflect.InvocationHandler;
2.
3. import java.lang.reflect.Proxy;
4.
5. public class Client
6.
7. {
8.
9.     public static void main(String[] args)
10.
11. {
12.
13. RealSubject realSubject =         RealSubject();
14.
15. InvocationHandler handler =         DynamicSubject(realSubject);
16.
17. Class<?> classType =         .getClass();
18.
19. // 下面的代码一次性生成代理
20.
21. Subject subject = (Subject) Proxy.newProxyInstance(classType
22.
23. .getClassLoader(), realSubject.getClass().getInterfaces(),
24.
25. handler);
26.
27. subject.request();
```

```

28.
29. System.out.println(subject.getClass());
30.
31.}
32.
33.}

```

接下来让我们来了解一下 Java 动态代理机制 Proxy 的构造方法：

清单 6. Proxy 构造方法

[\[html\] view plaincopyprint?](#)

```

1. // 由于 Proxy 内部从不直接调用构造函数，所以 private 类型意味着禁止任何调用
2.
3. private Proxy() {}
4.
5.
6. // 由于 Proxy 内部从不直接调用构造函数，所以 protected 意味着只有子类可以调用
7.
8. protected Proxy(InvocationHandler h) {this.h = h;}

```

接着，可以快速浏览一下 newProxyInstance 方法，因为其相当简单：

清单 7. Proxy 静态方法 newProxyInstance

[\[html\] view plaincopyprint?](#)

```

1. public static Object newProxyInstance(ClassLoader loader,
2.
3.      Class<?>[] interfaces,
4.
5.      InvocationHandler h)
6.

```

```

7.         throws IllegalArgumentException {
8.
9.
10.
11.    // 检查 h 不为空，否则抛异常
12.
13.    if (h == null) {
14.
15.        throw new NullPointerException();
16.
17.    }
18.
19.    // 获得与制定类装载器和一组接口相关的代理类类型对象
20.
21.    Class cl =                (loader, interfaces);
22.
23.
24.    // 通过反射获取构造函数对象并生成代理类实例
25.
26.    try {
27.
28.        Constructor cons = .getConstructor(constructorParams);
29.
30.        return (Object) cons.newInstance(new Object[] { h });
31.
32.    } catch (NoSuchMethodException e) { throw new InternalError(e.toString());
33.
34.    } catch (IllegalAccessException e) { throw new InternalError(e.toString());
35.
36.    } catch (InstantiationException e) { throw new InternalError(e.toString());
37.
38.    } catch (InvocationTargetException e) { throw new InternalError(e.toString());
39.
40.    }
41.}

```

由此可见，动态代理真正的关键是在 **getProxyClass** 方法，该方法负责为一组接口动态地生成代理类类型对象。

有很多条理由，人们可以否定对 **class** 代理的必要性，但是同样有一些理由，相信支持 **class** 动态代理会更美好。接口和类的划分，本就不是很明显，只是到了 **Java** 中才变得如此的细化。如果只从方法的声明及是否被定义来考量，有一种两者的混合体，它的名字叫抽象类。实现对抽象类的动态代理，相信也有其内在的价值。此外，还有一些历史遗留的类，它们将因为没有实现任何接口而从此与动态代理永世无缘。如此种种，不得不说是一个小小的遗憾。

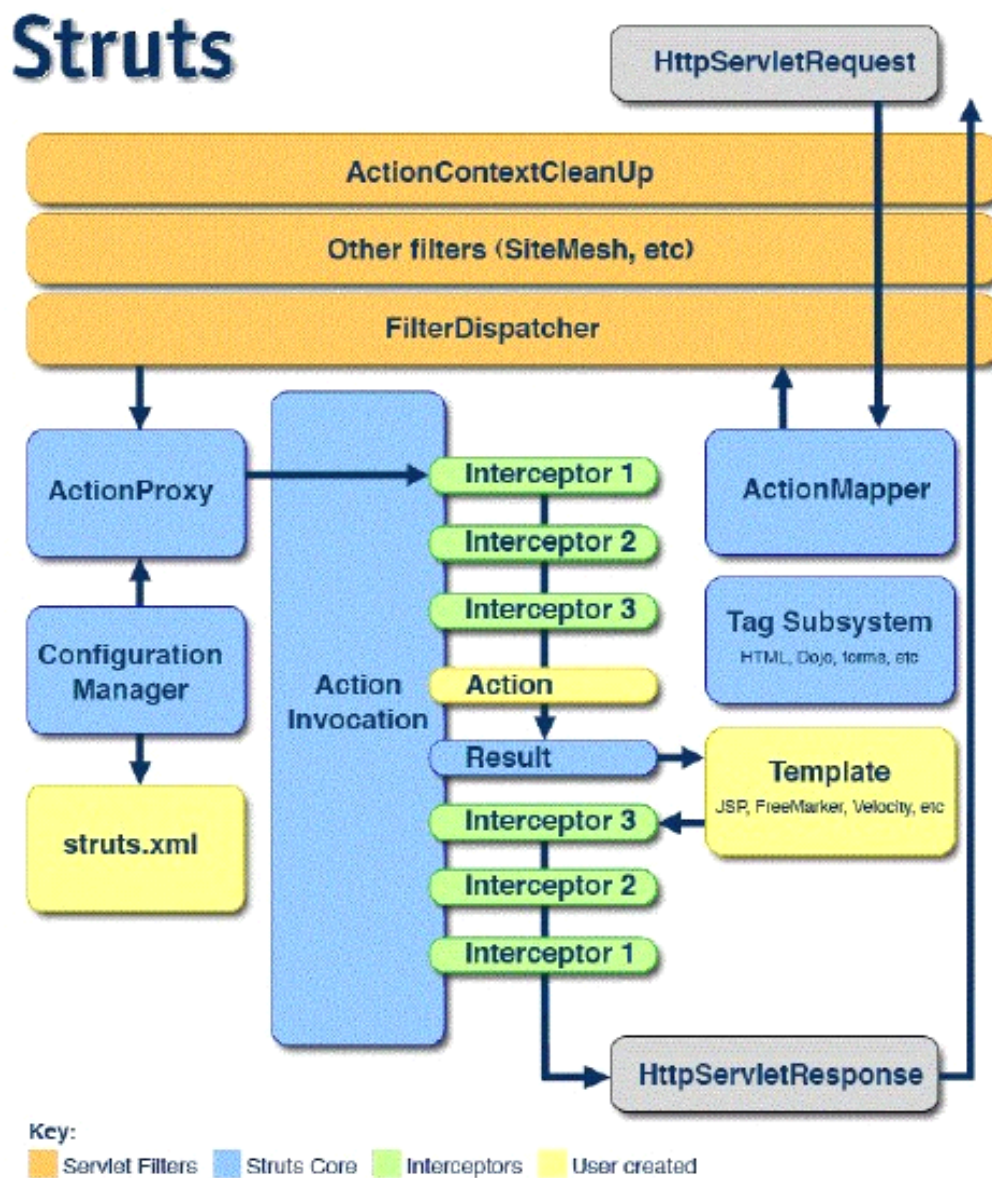
但是，不完美并不等于不伟大，伟大是一种本质，**Java** 动态代理就是佐例。

（四十）细谈 struts2（四）struts2 中 action 执行流程和源码分析

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

首先我们看一下 struts 官方给我们提供的 **struts** 执行流程



从上面流程图我们可以看出 **struts** 执行的流程大体分一下阶段：

1. 初始的请求通过一条标准的过滤器链，到达 `servlet` 容器(比如 `tomcat` 容器，`WebSphere` 容器)。
2. 过滤器链包括可选的 `ActionContextCleanUp` 过滤器，用于系统整合技术，如 `SiteMesh` 插件。
3. 接着调用 **`FilterDispatcher`**，**`FilterDispatcher`** 查找 **`ActionMapper`**，以确定这个请求是否需要调用某个 **`Action`**。
4. 如果 **`ActionMapper`** 确定需要调用某个 **`Action`**，**`FilterDispatcher`** 将控制权交给 **`ActionProxy`**。
5. **`ActionProxy`** 依照框架的配置文件(**`struts.xml`**)，找到需要调用的 **`Action`** 类。
6. **`ActionProxy`** 创建一个 **`ActionInvocation`** 的实例。**`ActionInvocation`** 先调用相关的拦截器（**`Action`** 调用之前的部分），最后调用 **`Action`**。
7. 一旦 **`Action`** 调用返回结果，**`ActionInvocation`** 根据 **`struts.xml`** 配置文件，查找对应的转发路径。返回结果通常是（但不总是，也可能是另外的一个 **`Action`** 链）**`JSP`** 技术或者 **`FreeMarker`** 的模版技术的网页呈现。**`Struts2`** 的标签和其他视图层组件，帮助呈现我们所需要的显示结果。在此，我想说清楚一些，最终的显示结果一定是 **`HTML`** 标签。标签库技术和其他视图层技术只是为了动态生成 **`HTML`** 标签。
8. 接着按照相反次序执行拦截器链(执行 **`Action`** 调用之后的部分)。最后，响应通过过滤器链返回（过滤器技术执行流程与拦截器一样，都是先执行前面部分，后执行后面部）。如果过滤器链中存在 **`ActionContextCleanUp`**,

FilterDispatcher 不会清理线程局部的 ActionContext。如果不存在
ActionContextCleanUp 过滤器，FilterDispatcher 会清除所有线程局部变量。

下面我们就来具体分析一下 **3-6** 四个步骤：

步骤三：FilterDispatcher 查找 ActionMapper，以确定这个请求是否需要调用某个 Action。

1)

[java] [view plaincopyprint?](#)

```
1. ActionMapping mapping;  
2.  
3.     try {  
4.  
5.         mapping = actionMapper.GetMapping(request, dispatcher.getConfigurati  
onManager());  
6.  
7.     } catch (Exception ex) {  
8.  
9.         log.error("error getting ActionMapping", ex);  
10.  
11.         dispatcher.sendError(request, response, servletContext, HttpServletRes  
ponse.SC_INTERNAL_SERVER_ERROR, ex);  
12.  
13.         return;  
14.     }
```

2) 调用 actionmapper 去寻找对应的 ActionMapping，因为 actionmapper 是一个接口，所有我们去他对应的实现类（DefaultActionMapper）里面去找 getMapping 方法，下面我们来看一下实现类里面的 getMapping 方法源代码：

[java] [view plain](#)[copy](#)[print](#)?

```
1. public ActionMapping getMapping(HttpServletRequest request,
2.
3.             ConfigurationManager configManager) {
4.
5.     ActionMapping mapping = new ActionMapping();
6.
7.     String uri = getUri(request);
8.     int indexOfSemicolon = uri.indexOf(";");
9.
10.    uri = (indexOfSemicolon > -1) ? uri.substring(0, indexOfSemicolon) : uri;
11.
12.    uri = dropExtension(uri, mapping);
13.
14.    if (uri == null) {
15.
16.        return null;
17.
18.    }
19.
20.    parseNameAndNamespace(uri, mapping, configManager);
21.
22.    handleSpecialParameters(request, mapping);
23.
24.    if (mapping.getName() == null) {
25.
26.        return null;
27.
28.    }
29.    parseActionName(mapping);
30.
31.    return mapping;
32.}
```


ActionMapping 代表 struts.xml 文件中的一个 Action 配置，被传入到 serviceAction 中。注意 ActionMapping 不代表 Action 集合，只代表某个对应的 Action。如果是一个 Action 请求，(请求路径在 struts.xml 有对应的 Action 配置，即 actionmapping 不为空)，则调用 dispatcher.serviceAction() 处理。找到对应的 ActionMapping，下一步就去找具体的执行哪一个 action，从 **FilterDispatcher** 源码中我们可以找到下一步流程：

[java] [view plaincopyprint?](#)

```
1. dispatcher.serviceAction(request, response, servletContext, mapping);
```

从上面可以看出，FilterDispatcher 类中是调用的 serviceAction 方法来寻找的去调用哪一个 action。serviceAction()方法作用：加载 Action 类，调用 Action 类的方法，转向到响应结果。响应结果指<result/> 标签所代表的对象。

步骤四、五、六：如果 ActionMapper 确定需要调用某个 Action，FilterDispatcher 将控制权交给 ActionProxy。

我们来看一下具体的 serviceAction 源码：

[java] [view plaincopyprint?](#)

```
1. public void serviceAction(HttpServletRequest request, HttpServletResponse response,  
    nse,  
2.  
3. ServletContext context, ActionMapping mapping) throws ServletException {  
4.  
5. Map<String, Object> extraContext = createContextMap  
6.
```

```
7. (request, response, mapping, context);
8.
9. //1 以下代码目的为获取 ValueStack，代理在调用的时候使用的是本值栈的副本
10.
11.ValueStack stack = (ValueStack) request.getAttribute
12.
13.(ServletActionContext.STRUTS_VALUESTACK_KEY);
14.
15.boolean nullStack = stack == null;
16.
17.if (nullStack) {
18.
19.ActionContext ctx = ActionContext.getContext();
20.
21.if (ctx != null) {
22.
23.stack = ctx.getValueStack();
24.
25.}
26.
27.}
28.
29.//2 创建 ValueStack 的副本
30.
31.if (stack != null) {
32.
33.extraContext.put(ActionContext.VALUE_STACK,
34.
35.valueStackFactory.createValueStack(stack));
36.
37.}
38.
39.String timerKey = "Handling request from Dispatcher";
40.
41.try {
```

```
42.
43.UtilTimerStack.push(timerKey);
44.
45.//3 这个是获取配置文件中<action/> 配置的字符串，action 对象已经在核心控制器中
    创建
46.
47.String namespace = mapping.getNamespace();
48.
49.String name = mapping.getName();
50.
51.String method = mapping.getMethod();
52.
53.// xwork 的配置信息
54.
55.Configuration config = configurationManager.getConfiguration();
56.
57.//4 动态创建 ActionProxy
58.
59.ActionProxy proxy =
60.
61.config.getContainer().getInstance(ActionProxyFactory.class).
62.
63.createActionProxy(namespace, name, method, extraContext, true, false);
64.
65.request.setAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY,
66.
67.proxy.getInvocation().getStack());
68.
69.//5 调用代理
70.
71.if (mapping.getResult() != null) {
72.
73.Result result = mapping.getResult();
74.
75.result.execute(proxy.getInvocation());
```

```
76.
77.} else {
78.
79.proxy.execute();
80.
81.}
82.
83.//6 处理结束后，恢复值栈的代理调用前状态
84.
85.if (!nullStack) {
86.
87.request.setAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY, stack);
88.
89.}
90.
91.} catch (ConfigurationException e) {
92.
93.//7 如果 action 或者 result 没有找到，调用 sendError 报 404 错误
94.
95.}
```

关于 **valuestack** 说明一下：

1.valueStack 的建立是在 doFilter 的开始部分，在 Action 处理之前。即使访问静态资源 ValueStack 依然会建立，保存在 request 作用域。

2. ValueStack 在逻辑上包含 2 个部分：object stack 和 context map，object stack 包含 Action 与 Action 相关的对象。

context map 包含各种映射关系。

request,session,application,attr,parameters 都保存在 context map 里。

parameters: 请求参数

attr: 依次搜索 page, request, session, 最后 application 作用域。

几点说明：

1. Valuestack 对象保存在 request 里，对应的 key 是 `ServletContext.STRUTS_VALUESTACK_KEY`。调用代理之前首先创建 Valuestack 副本，调用代理时使用副本，调用后使用原实例恢复。本处的值栈指 object stack。

2. Dispatcher 实例，创建一个 Action 代理对象。并把处理委托给代理对象的 `execute` 方法。

最后我们在一起看一下 `ActionInvocation` 实现类中 `invoke` 方法执行的流程：

invoke 源代码：

[java] [view plaincopyprint?](#)

```
1. public String invoke() throws Exception {
2.
3.     String profileKey = "invoke: ";
4.
5.     try {
6.
7.         UtilTimerStack.push(profileKey);
8.
9.         if (executed) {
10.
11.             throw new IllegalStateException("Action has already executed");
12.
13.         }
14.
15.         if (interceptors.hasNext()) {
16.
17.             final InterceptorMapping interceptor = (InterceptorMapping) interceptors.
                next();
18.
```

```

19.         String interceptorMsg = "interceptor: " + interceptor.getName();
20.
21.         UtilTimerStack.push(interceptorMsg);
22.
23.         try {
24.
25.             resultCode = interceptor.getInterceptor().intercept(DefaultActi
onInvocation.this);
26.
27.         }
28.
29.         finally {
30.
31.             UtilTimerStack.pop(interceptorMsg);
32.
33.         }
34.
35.     } else {
36.
37.         resultCode = invokeActionOnly();
38.
39.     }
40.
41.
42.     if (!executed) {
43.
44.         if (preResultListeners != null) {
45.
46.             for (Object preResultListener : preResultListeners) {
47.
48.                 PreResultListener listener = (PreResultListener) preResultListener;
49.
50.
51.                 String _profileKey = "preResultListener: ";

```

```
52.
53.         try {
54.
55.             UtilTimerStack.push(_profileKey);
56.
57.             listener.beforeResult(this, resultCode);
58.
59.         }
60.
61.         finally {
62.
63.             UtilTimerStack.pop(_profileKey);
64.
65.         }
66.
67.     }
68.
69. }
70.
71.
72.     if (proxy.getExecuteResult()) {
73.
74.         executeResult();
75.
76.     }
77.
78.
79.     executed = true;
80.
81. }
82.     return resultCode;
83.
84. }
85.
86.     finally {
```

```
87.  
88.         UtilTimerStack.pop(profileKey);  
89.  
90.     }  
91.}
```

这里算是执行 **action** 中方法的最后一步了吧，至此，**action** 的整个流程就基本差不多了，从头到尾看下来，说实话，感触很多，很多不明白的地方，这算是近了自己最大的努力去看这些源码，感觉从里面收获了很多，里面很多的机制和知识点值得我们去学习，记住了圣思源张龙老师的那句话：**源码面前，一目了然**

（四十一）细谈 struts2（五）action 基础知识和数据校验

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

一：首先看一下 **struts2** 中 **action** 的实现方式：

1.建立普通的 pojo 类：这种方式能够实现简单的 action 功能，但 struts2 内自带的一些验证和其他功能不能够实现

2.继承 ActionSupport 类实现 action，因为 ActionSupport 已经实现了 Action 接口，还实现了 Validateable 接口，提供了数据校验功能。通过继承该 ActionSupport 类，可以简化 Struts 2 的 Action 开发。

3.实现 action 接口，这个接口里面定义了一些 action 所要实现的功能的标准，但验证等功能没有，所以一般还是继承 actionsupport 来实现 action

Action 跟 Actionsupport 的区别：

当我们在写 action 的时候,可以实现 Action 接口,也可以继承 Actionsupport 这个类.到底这两个有什么区别呢？

Action 接口有：

[java] [view plaincopyprint?](#)

```
1. public static final java.lang.String SUCCESS = "success";
2.
3. public static final java.lang.String NONE = "none";
4.
5. public static final java.lang.String ERROR = "error";
6.
7. public static final java.lang.String INPUT = "input";
8.
9. public static final java.lang.String LOGIN = "login";
```

10.

11. `public abstract java.lang.String execute() throws java.lang.Exception;`

而 `ActionSupport` 这个工具类在实现了 `Action` 接口的基础上还定义了一个 `validate()` 方法, 重写该方法, 它会在 `execute()` 方法之前执行, 如校验失败, 会转入 `input` 处, 必须在配置该 `Action` 时配置 `input` 属性。

另外, `ActionSupport` 还提供了一个 `getText(String key)` 方法还实现国际化, 该方法从资源文件上获取国际化信息。

这样在自定义标签时可以定义一个变量为 `new actionsupport` 对象实现国际化。

ActionSupport 类的作用

`struts2` 不要求我们自己设计的 `action` 类继承任何的 `struts` 基类或 `struts` 接口, 但是我们为了方便实现我们自己的 `action`, 大多数情况下都会继承 `com.opensymphony.xwork2.ActionSupport` 类, 并重写此类里的 `public String execute() throws Exception` 方法。因为此类中实现了很多的实用借口, 提供了很多默认方法, 这些默认方法包括国际化信息的方法、默认的处理用户请求的方法等, 这样可以大大的简化 `Action` 的开发。

`Struts2` 中通常直接使用 `Action` 来封装 HTTP 请求参数, 因此, `Action` 类里还应该包含与请求参数对应的属性, 并且为属性提供对应的 `getter` 和 `setter` 方法。

二. action 数据校验

在上面应用中, 即使浏览者输入任何用户名、密码, 系统也会处理用户请求。在我们整个 `HelloWorld` 应用中, 这种空用户名、空密码的情况不会引起太大的问题。但如果数据需要保存到数据库, 或者需要根据用户输入的用户

名、密码查询数据，这些空输入可能引起异常。为了避免用户的输入引起底层异常，通常我们会在进行业务逻辑操作之前，先执行基本的数据校验。

Action 数据校验功能是 **struts2** 给我们提供的一个服务器端简单验证的功能，这个功能使我们简化了一些没必要的代码。下面看一下具体实现：

1.继承 **ActionSupport**

ActionSupport 类是一个工具类，它已经实现了 **Action** 接口。除此之外，它还实现了 **Validateable** 接口，提供了数据校验功能。通过继承该 **ActionSupport** 类，可以简化 Struts 2 的 **Action** 并在 **Validateable** 接口中定义了一个 **validate()** 方法，重写该方法，如果校验表单输入域出现错误，则将错误添加到 **ActionSupport** 类的 **fieldErrors** 域中，然后通过 **OGNL** 表达式负责输出为了让 Struts 2 增加输入数据校验的功能，改写程序中的 **LoginAction**，增加重写 **validate** 方法。下面看一下具体代码实现：

[java] [view plaincopyprint?](#)

```
1. package com.bzu.action;
2.
3. import com.opensymphony.xwork2.ActionSupport;
4.
5. public class LoginAction extends ActionSupport {
6.
7.     private String username;
8.
9.     private String password;
10.
11. public String getUsername() {
12.
13. return username;
14.
```

```
15.}
16.
17.public void setUsername(String username) {
18.
19.this.username = username;
20.
21.}
22.
23.public String getPassword() {
24.
25.return password;
26.
27.}
28.
29.public void setPassword(String password) {
30.
31.this.password = password;
32.
33.}
34.
35.public void validate() {
36.
37.if("").equals(username))
38.
39.this.addActionError("soory,the username can't blank");
40.
41.if("").equals(password))
42.
43.this.addActionError("soory,the password can't blank");
44.
45.}
46.
47.public String execute(){
48.
49.if(username.equals("admin")&&password.equals("123456"))
```

```
50.  
51.return "success";  
52.  
53.return "fail";  
54.  
55.}  
56.  
57.}
```

这里简单的实现了表单数据验证功能，上面的 Action 类重写了 validate 方法，该方法会在执行系统的 execute 方法之前执行，如果执行该方法之后，Action 类的 fieldErrors 中已经包含了数据校验错误，请求将被转发到 input 逻辑视图处。但**还是要注意以下几点：**

1.在实现表单验证功能的时候一定不要忘了在 struts.xml 中相对应的 action 中配置 result= “input”，因为表单验证失败默认返回的字符串为 input，如果没有的话会找不到这个结果而报错。

2.数据验证中，如果数据不符的时候可以报三种错误，我们上面代码中只是列举了 action 错误，另外两种是 Field 字段的错误，还有一种就是 actionMessage。

3.注意在显示界面接收 action 错误时，要在想显示错误的地方加上 `<s:actionerror/>` 标签，如果想接收 Filed 域的错误时，一定要用 struts 标签，如果不用的话是不会显示字段错误的

2.使用 Struts 2 的校验框架

上面的输入校验是通过重写 ActionSupport 类的 validate 方法实现的，这种方法虽然不错，但需要大量重写的 validate 方法——毕竟，重复书写相同的代码不是一件吸引人的事情。

类似于 Struts 1，Struts 2 也允许通过定义配置文件来完成数据校验。

Struts 2 的校验框架实际上是基于 XWork 的 validator 框架。

下面还是使用原来的 Action 类（即不重写 validate 方法），却增加一个校验配置文件，校验配置文件通过使用 Struts 2 已有的校验器，完成对表单域的校验。Struts 2 提供了大量的数据校验器，包括表单域校验器和非表单域校验器两种。本应用主要使用了 requiredstring 校验器，该校验器是一个必填校验器——指定某个表单域必须输入。

下面是校验规则的定义文件：

[html] view plaincopyprint?

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2/
   /EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
4.
5. <validators>
6.
7. <field name="username" type="requiredstring">
8.
9. <field-validator type="requiredstring">
10.
11. <param name="required" value="true"/>false</param>
12.
13. <message>username can't be blank!</message>
14.
15. </field-validator>
16.
17. <field-validator type="requiredstring">
18.
19. <param name="required" value="true"/>4</param>
20.
21. <param name="required" value="true"/>6</param>
```

```
22.
23. <param name=         >false</param>
24.
25. <message key=         ></message>
26.
27. </field-validator>
28.
29. </field>
30.
31. <field name=         >
32.
33. <field-validator type=         >
34.
35. <message>password can't be blank!</message>
36.
37. </field-validator>
38.
39. <field-validator type=         >
40.
41. <param name=         >4</param>
42.
43. <param name=         >6</param>
44.
45. <message>length of password should be between ${minLength} and ${maxLength}
    h}</message>
46.
47. </field-validator>
48.
49. </field>
50.
51. <field name=         >
52.
53. <field-validator type=         >
54.
55. <message>age can't be blank!</message>
```

```
56.
57. </field-validator>
58.
59. <field-validator type="range">
60.
61. <param name="min">10</param>
62.
63. <param name="max">40</param>
64.
65. <message>age should be between ${min} and ${max}</message>
66.
67. </field-validator>
68.
69. </field>
70.
71. </validators>
```

定义完该校验规则文件后，该文件的命名应该遵守如下规则：

ActionName-validation.xml：其中 ActionName 就是需要校验的 Action 的类名。因此上面的校验规则文件应该命名为“LoginAction-validation.xml”，且该文件应该与 Action 类的 class 文件位于同一个路径下。因此，将上面的校验规则文件放在 WEB-INF/classes/lee 路径下即可。当然，在 struts.xml 文件的 Action 定义中，一样需要定义 input 的逻辑视图名，将 input 逻辑视图映射到 login.jsp 页面。在这种校验方式下，无需书写校验代码，只需要通过配置文件指定校验规则即可，因此提供了更好的可维护性。

三、action 中的执行方法

Action 中默认的执行方法是 execute 方法，这个方法执行请求，然后转向其他的页面，这是常规的做法，但有时候我们不想用这个方法名，为了代码的

可读性，我们希望让他执行我们自己定义的方法，下面我们就来看一下执行其他方法的两种方法：

1.在 **struts.xml** 配置 **method** 属性

其实执行 **execute** 方法是对应 **action** 在配置文件 **method** 的默认方法，所以要想执行其他的方法，我们可以修改这里的默认方法，只要把默认的方法改为我们自定义的方法就可以了。部分配置代码：

[java] [view plaincopyprint?](#)

```
1. <action name="LoginAction" class="com.bzu.action.LoginAction" method="login">
2.
3. <result name="success">success.jsp</result>
4.
5. <result name="fail">fail.jsp</result>
6.
7. <result name="input">login.jsp</result>
8.
9. </action>
10.
11.<action name="RegisteAction" class="com.bzu.action.LoginAction" method="regis
    te">
12.
13.<result name="success">success.jsp</result>
14.
15.<result name="fail">fail.jsp</result>
16.
17.<result name="input">login.jsp</result>
18.
19.</action>
```

2.DMI（动态直接调用）这种方法，不需要进行 struts.xml 的配置。而是在 html 或者 jsp 页面中通过标示符号指定了要调用的方法。 关键的标示符号为"! "号，具体看一下下面表单：

[java] view plaincopyprint?

```
1. <s:form action="LoginAction!login">
2.
3.   <s:actionerror/>
4.
5.   username: <s:textfield name="username"/></s:textfield>
6.
7.   password: <s:password name="password"/></s:password>
8.
9.   <s:submit value="提交"/></s:submit>
10.
11. </s:form>
```

3.提交按钮指定提交方法，普通的提交按钮我们会这么

写： `<s:submit value="提交"/></s:submit>`

当我们想提交到我们指定的方法时我们可以在这个标签里添加一个 method 属性指定要提交的方法，如下：

[html] view plaincopyprint?

```
1. <s:submit value=      method=      ></s:submit>
```

4.使用通配符配置 Action，这种方法可以解决 action 配置文件混乱的问题，减少 action 的配置：

[html] view plaincopyprint?

```

1. <action name=          class=          method=      >
2.
3.     <result name=          >/WEB-INF/jsp/{1}_success.jsp</result>
4.
5. </action>
6.
7. <SPAN style=          ><SPAN style=          > </SPAN>
   </SPAN>

```

在 `name` 属性的值后面加上`*`，意思是只要你请求的 `action` 名字以 `helloworld` 开头，本 `action` 就是你找的 `action`，然后 `method` 是大括号加上 `1`，这个 `1` 代表第一个星号的值，这样就可以动态执行请求的方法。

最后说一点，有时候用户在地址栏随便输入的时候，找不到对应的 `action`，直接对报出一些错误，这样的界面一般都很难看，所以为了能给用户一个友好的提示界面，一般我们会再 `struts.xml` 文件配置默认的 `action`，代码如下：

```

<struts>
  <package name="default" extends="struts-default">
    <default-action-ref name="defaultAction"/ >
    <action name="defaultAction">
      <result>/index.jsp</result>
    </action>
  </package>
</struts>

```

如果请求的Action不存在，将转发到default.jsp

省略class属性，将使用 Action Support类

（四十二）大话设计模式（八）状态模式

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。看起来，状态模式好像是神通广大很厉害似的——居然能够“修改自身的类”！下面让我们一起来看一下他的厉害吧！

适用场景：

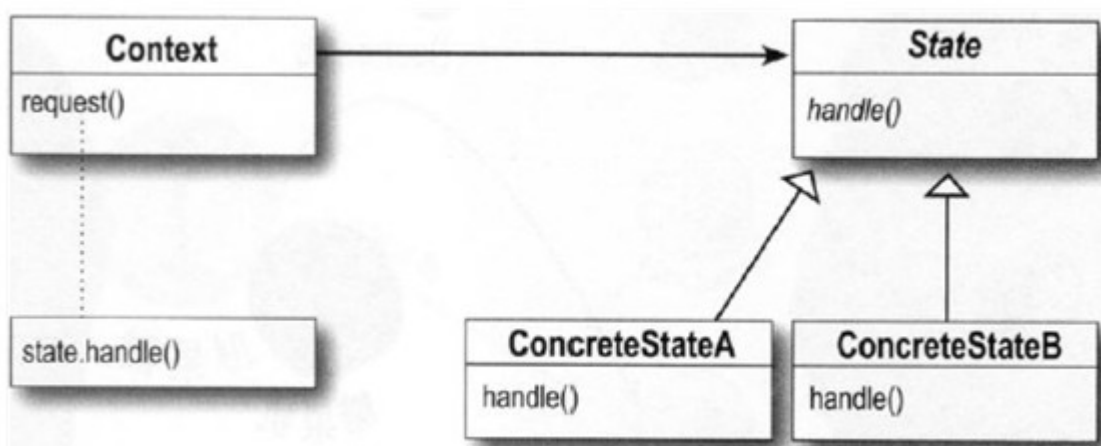
状态模式主要解决的是当控制一个对象状态装换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简单化。当一个对象行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。

要点：

1. 策略模式和状态模式是双胞胎，它们有相同的类图，但是它们的意图不同。策略模式是围绕可以互换的算法来成功创建业务的,然而状态模式是通过改变对象内部的状态来帮助对象控制自己的行为。
2. Context 将与状态相关的操作委托给当前的 Concrete State 对象处理。
3. Context 可将自身作为一个参数传递给处理该请求的状态对象。这使得状态对象在必要时可访问 Context。

4. **Context** 或 **Concrete State** 类都可决定哪个状态是另外哪一个的后继者，以及是在何种条件下进行状态转换。也就是说可以在 **State** 中保存对 **Concrete State** 的引用，在必要时设置具体的状态，做到状态的转换。

5. 一般来讲，当状态转换是固定的时候，状态转换就适合放在 **Context** 中。然而，当转换是更动态的时候，通常会放到具体的状态类中进行。（具体状态类持有 **Context** 的引用，实现状态的转换）



Context 类：维护一个 **ConcreteState** 子类的一个实例，这个实例定义当前的状态。

State 类：抽象状态类，定义一个接口以封装与 **Context** 的一个特定状态相关的行为。

ConcreteStateA, ConcreteStateB 类：具体状态类，每一个子类实现一个与 **Context** 的一个状态相关的行为。

具体实现：

先定义接口 *UserState*

[java] [view plaincopyprint?](#)

```
1. 1 public interface UserState {
2.
```

```
3. 2 public UserService userService = new UserService();
4.
5. 3 // 状态为 0 代表普通用户,这些状态主要用于持久化到数据
6.
7. 4 public final static int COMMON_STATUS = 0;
8.
9. 5 // 状态为 1 代表待审核用户
10.
11.6 public final static int FOR_VERIFY_STATUS = 1;
12.
13.7 // 状态为 2 是代表 vip 用户
14.
15.8 public final static int VIP_STATUS = 2;
16.
17.9
18.
19.10 // 提交信息
20.
21.11 public void submit() throws Exception;
22.
23.12
24.
25.13 // 审核没通过
26.
27.14 public void rollback() throws Exception;
28.
29.15
30.
31.16 // 审核通过
32.
33.17 public void pass() throws Exception;
34.
35.18
36.
37.19 public int getStatus();
```

代表各个状态的类

未提交信息的用户 *NotSubmitUser.java*

[java] [view plain](#)[copy](#)[print?](#)

```
1. 2 public class NotSubmitUser implements UserState {
2.
3. 3   public void pass() throws Exception {
4.
5. 4       throw new Exception("no support method");
6.
7. 5   }
8.
9. 6
10.
11.7   public void rollback() throws Exception {
12.
13.8       throw new Exception("no support method");
14.
15.9   }
16.
17.10
18.
19.11   public void submit() throws Exception {
20.
21.12       User user = userService.getUser(0);
22.
23.13       user.setStaust(FOR_VERIFY_STATUS);
24.
25.14       userService.update(user);
26.
27.15   }
```



```

28.
29.16
30.
31.17 public int getStatus() {
32.
33.18     return COMMON_STATUS;
34.
35.19 }
36.
37.20 }

```

待审核的用户 *WaitForVerifyUser.java*

Java 代码

[java] [view plain](#)[copy](#)[print?](#)

```

1. 22 public class WaitForVerifyUser implements UserState {
2.
3. 23
4.
5. 24 public void pass() throws Exception {
6.
7. 25     User user = userService.getUser(0);
8.
9. 26     user.setStauts(VIP_STATUS);
10.
11.27     userService.update(user);
12.
13.28 }
14.
15.29

```

```
16.  
17.30 public void rollback() throws Exception {  
18.  
19.31     User user = userService.getUser(0);  
20.  
21.32     user.setStauts(COMMON_STATUS);  
22.  
23.33     userService.update(user);  
24.  
25.34 }  
26.  
27.35  
28.  
29.36 public void submit() throws Exception {  
30.  
31.37     throw new Exception("no support method");  
32.  
33.38 }  
34.  
35.39  
36.  
37.40 public int getStatus() {  
38.  
39.41     return VIP_STATUS;  
40.  
41.42 }  
42.  
43.43 }
```

审核通过的用户, vip VipUser.java

[java] [view plaincopyprint?](#)

```

1. 45 public class VipUser implements UserState {
2.
3. 46
4.
5. 47     public void pass() throws Exception {
6.
7. 48         throw new Exception("no support method");
8.
9. 49     }
10.
11.50
12.
13.51     public void rollback() throws Exception {
14.
15.52         User user = userService.getUser(0);
16.
17.53         user.setStauts(COMMON_STATUS);
18.
19.54         userService.update(user);
20.
21.55     }
22.
23.56
24.
25.57     public void submit() throws Exception {
26.
27.58         throw new Exception("no support method");
28.
29.59     }
30.
31.60
32.
33.61     public int getStatus() {
34.
35.62         return FOR_VERIFY_STATUS;

```

```
36.  
37.63 }  
38.  
39.64 }
```

根据用户的 *status* 的返回不同的 *UserState*

[java] [view plain](#)[copy](#)[print?](#)

```
1. 66 public class UserStateFactory {  
2.  
3. 67     static List list = null;  
4.  
5. 68     static {  
6.  
7. 69         list = new ArrayList();  
8.  
9. 70         list.add(new NotSubmitUser());  
10.  
11.71         list.add(new VipUser());  
12.  
13.72         list.add(new WaitForVerifyUser());  
14.  
15.73  
16.  
17.74     }  
18.  
19.75  
20.  
21.76     public static UserState getUserState(int status) {  
22.  
23.77         for (UserState state : list) {  
24.
```

```

25.78         if (state.getStatus() == status)
26.
27.79         return state;
28.
29.80     }
30.
31.81     return null;
32.
33.82 }
34.
35.83 }

```

状态模式操作是固定的，但是接受者类不相同。多态性的原则实际执行哪个方法不仅取决于方法签名，还取决于操作的接受者类。

该例子只是 *state* 模式的一个场景的应用，比较具体。

状态模式的优点：

1.非常好的扩展性---假设增加一个用户组：当 *vip* 用户的信用达到一定程度后，升级到永久的

vip。这时只需另外添加一个状态类，对原来的代码并不需要做改动。

2.代码结构清晰，不易出错。即程序健壮--除 *UserStateFactory* 的 *getUserState(int status)*

外，其他方法的逻辑非常的简单，都不包含局部变量。如果程序不需要持久化到数据库，也不需要

getUserState(int status)，该方法是状态模式内容之外的。用状态模式类的数量会大大的增

加。

说明：该例子只是状态模式的一个应用，并不是状态模式，是属于比较具体

（四十三）细谈 struts2（六）获取 servletAPI 和封装表单数据

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

一：获取 **servletAPI** 的三种方法

在传统的 Web 开发中，经常会用到 Servlet API 中的 `HttpServletRequest`、`HttpSession` 和 `ServletContext`。Struts 2 框架让我们可以直接访问和设置 `action` 及模型对象的数据，这降低了对 `HttpServletRequest` 对象的使用需求，同时降低了对 `servletAPI` 的依赖性，从而降低了与 `servletAPI` 的耦合度。但在某些应用中，我们可能会需要在 `action` 中去访问 `HttpServletRequest` 等对象，所以有时候我们不得不拉近 `struts2` 和 `servletAPI` 的关系，但 `struts2` 也有尽量减少耦合度的方法，下面我们就一起具体看一下在 `struts2` 中获得 `ServletAPI` 的三种方法：

1. `ServletAPI` 解藕方式（一）获取 `Map` 对象：

为了避免与 `Servlet API` 耦合在一起，方便 `Action` 类做单元测试，Struts 2 对 `HttpServletRequest`、`HttpSession` 和 `ServletContext` 进行了封装，构造了三个 `Map` 对象来替代这三种对象，在 `Action` 中，直接使用 `HttpServletRequest`、`HttpSession` 和 `ServletContext` 对应的 `Map` 对象来保存和读取数据。可以通过 `com.opensymphony.xwork2.ActionContext` 类来

得到这三个对象。**ActionContext** 是 **Action** 执行的上下文，保存了很多对象如 **parameters**、**request**、**session**、**application** 和 **locale** 等。通过 **ActionContext** 类获取 **Map** 对象的方法为：

[Java] [view plaincopyprint?](#)

1. `ActionContext context = ActionContext.getContext();` --得到 **Action** 执行的上下文
- 2.
3. `Map request = (Map) context.get("request");` --得到 **HttpServletRequest** 的 **Map** 对象
- 4.
5. `Map session = context.getSession();` --得到 **HttpSession** 的 **Map** 对象
- 6.
7. `Map application = context.getApplication();` --得到 **ServletContext** 的 **Map** 对象

ActionContext 中保存的数据能够从请求对象中得到，其中的奥妙就在于 **Struts 2** 中的 `org.apache.struts2.dispatcher.StrutsRequestWrapper` 类，这个类是 **HttpServletRequest** 的包装类，它重写了 `getAttribute()` 方法（在页面中获取 **request** 对象的属性就要调用这个方法），在这个方法中，它首先在请求对象中查找属性，如果没有找到（如果你在 **ActionContext** 中保存数据，当然就找不到了），则到 **ActionContext** 中去查找。这就是为什么在 **ActionContext** 中保存的数据能够从请求对象中得到的原因。

2.IOC（控制反转）获取 **servletAPI**

Action 类还有另一种获得 **ServletAPI** 的解耦方式，这就是我们可以让他实现某些特定的接口，让 **Struts2** 框架在运行时向 **Action** 实例注入 **request**、**session** 和 **application** 对象。这种方式也就是 **IOC**（控制反转）方式，与之对应的三个接口和它们的方法如下所示：

[java] [view plaincopyprint?](#)

```
1. public class SampleAction implements Action,
2. RequestAware, SessionAware, ApplicationAware
3. {
4.     private Map request;
5.     private Map session;
6.     private Map application;
7.
8.     @Override
9.     public void setRequest(Map request)
10. {this.request = request;}
11.
12. @Override
13. public void setSession(Map session)
14. {this.session = session;}
15.
16. @Override
17. public void setApplication(Map application)
18. {this.application = application;}
19.
20. }
```

`ServletRequestAware` 接口和 `ServletContextAware` 接口不属于同一个包，前者在 `org.apache.struts2.interceptor` 包中，后者在 `org.apache.struts2.util` 包中，这很让人迷惑。

3. 与 **Servlet API** 耦合的访问方式

直接访问 **Servlet API** 将使你的 **Action** 与 **Servlet** 环境耦合在一起，我们知道对于 `HttpServletRequest`、`HttpServletResponse` 和 `ServletContext` 这些

对象，它们都是由 **Servlet** 容器来构造的，与这些对象绑定在一起，测试时就需要有 **Servlet** 容器，不便于 **Action** 的单元测试。但有时候，我们又确实需要直接访问这些对象，那么当然是以完成任务需求为主。要直接获取 **HttpServletRequest** 和 **ServletContext** 对象，可以使用 **org.apache.struts2.ServletActionContext** 类，该类是 **ActionContext** 的子类，在这个类中定义下面两个静态方法：

1.得到 **HttpServletRequest** 对象：

```
public static HttpServletRequest getRequest()
```

2.得到 **ServletContext** 对象：

```
public static ServletContext getServletContext()
```

此外，**ServletActionContext** 类还给出了获取 **HttpServletResponse** 对象的方法，如下：

```
public static HttpServletResponse getResponse()
```

ServletActionContext 类并没有给出直接得到 **HttpSession** 对象的方法，

HttpSession 对象可以通过 **HttpServletRequest** 对象来得到。

除了上述的方法调用得到 **HttpServletRequest** 和 **ServletContext** 对象外，还可以调用 **ActionContext** 对象的 **get()**方法，传递

ServletActionContext.HTTP_REQUEST 和

ServletActionContext.SERVLET_CONTEXT 键值来得到

HttpServletRequest 和 **ServletContext** 对象同样的，也可以向 **ActionContext** 的 **get()**方法传递 **ServletActionContext.HTTP_RESPONSE** 键值来得到

HttpServletResponse 对象

总结：通过上面三种方式的讲解我们可以看出，三种获得 `servletAPI` 的方式基本都差不多，通常我们建议大家采用第一种方式来获取 `HttpServletRequest` 和 `ServletContext` 对象，这样简单而又清晰，并且降低了和 `servletAPI` 的耦合度，这样也方便进行单元测试

二：struts2 封装请求参数三种方式

在 `struts2` 开发应用中，我们可能经常要求获得视图层传过来的很多数据，一般都是一个实体类的 `n` 多属性，很多时候实体类的属性特别多，这时候如果还是按以前的方式在 `action` 里面一个个的定义出这些属性的私有变量，然后在提供 `set`、`get` 方法的话，这样就会使整个 `action` 太臃肿，严重妨碍了代码的可阅读性，并且也违背了代码的可复用性，这时我们就需要对这些请求参数进行封装，提高代码的可复用性，下面我们就一起来具体看一下三种封装请求参数的方法：

1.利用实体类封装参数

这种方式是封装参数最简单的方法，一般也比较常用，因为在我们的 `struts` 应用程序中，我们一般会根据数据库的信息写出对应的实体类，所以这正好使我们可以利用的，下面我们看一下具体操作：

1.创建实体类 `user`（包括用户名和密码属性），这里比较简单，我们就不贴出代码了。

2.创建 `action`，这里我们主要是来看一下 `action` 接收数据的属性这个地方，我们就不是在一定义这些属性的私有变量了，我们直接定义一个对应实体类的私有对象就可以了，代码如下：

[\[java\] view plaincopyprint?](#)

```

1. package com.bzu.action;
2.
3. publicclass LoginAction extends ActionSupport {
4.
5.     private User user;
6.
7.     public User getUser() {
8.
9.         returnuser;
10.
11.     }
12.
13.     publicvoid setUser(User user) {
14.
15.         this.user = user;
16.
17.     }
18.
19.     public String execute(){
20.
21.         if(user.getUsername().equals("admin")&&user.getPassword().equals("123456"))
22.
23.             return"success";
24.
25.         return"fail";
26.
27.     }
28.
29.}

```

3.定义表单，这里我们需要注意一下，这里表单里面的控件的 **name** 属性定义有一定的要求，定义 **name** 时我们应该定义为：**对象.属性**的形式，示例代码：

[html] view plaincopyprint?

```
1. <s:form action=           >
2.
3. <s:actionerror/>
4.
5. <s:textfield name=           ></s:textfield>
6.
7. <s:password name=           ></s:password>
8.
9. <s:submit value=           ></s:submit>
10.
11. </s:form>
```

4.配置 **struts.xml**，这里配置和平常一样，这里就不再重复了

至此，我们简单的实体类封装请求参数就完成了，我相信大家一定也会感觉很简单吧

2.模型驱动封装请求参数

模型驱动是指使用 **JavaBean** 来封装来回请求的参数.这种方式的好处就是减少了 **action** 的压力。既用于封装来回请求的参数,也保护了控制逻辑,使它的结构清晰.这就是模型驱动的优势.

下面我们具体来看一下模型驱动的具体实现：

模型驱动的实现主要是体现在 **action** 上

- 1.首先建立一个实体，比较简单，这里就不再写了。
- 2.建立 **action** 类，继承自 **ActionSupport**，实现 **ModelDriven** 接口，这个接口定义了一个 **getModel()**方法，用于返回定义的 **Model**，然后调用 **set** 方法，进行赋值。代码示例：

[java] [view plain](#)[copy](#)[print?](#)

```
1. <SPAN xmlns="http://www.w3.org/1999/xhtml">publicclass LoginAction3 extends
    ActionSupport implementsModelDriven<User> {
2.     private User user=new User();//这里记住要实例化
3.     private LoginService loginService=new LoginServiceImpl();//这里是调用登录的业
        务处理逻辑
4.     @Override
5.     public User getModel() {
6.         //TODOAuto-generated method stub
7.         return user;
8.     }
9.     public String execute()
10.    {
11.        System.out.println(user.getUsername());
12.        System.out.println(user.getPassword());
13.
14.        if(loginService.isLogin(user.getUsername(),user.getPassword()))
15.        {
16.            return SUCCESS;
17.        }
18.        return INPUT;
19.    }
20.}</SPAN>
```

在 **com.opensymphony.xwork2.ModelDriven** 接口源代码中有一段很重要的说明，现抄录如下

ModelDriven Actions provide a model object to be pushed onto the ValueStack in addition to the Action itself, allowing a FormBean type approach like Struts

翻译：模型驱动的动作。将模型对象以及 Action 对象都放到 ValueStack 里面，允许像 Struts 一样的 FormBean 方式

也即：一个 **Action** 要想成为模型驱动的话，就必须实现 **ModelDriven** 接口，而我们之前所一直继承的 **ActionSupport** 类并没有实现 **ModelDriven** 接口

ModelDrivenAction 类的执行流程是：首先调用 getModel() 方法得到 User 对象，接着根据 JavaBean 的原则将客户端传过来的属性，一个一个的 set 到 User 对象的属性中，将属性全部 set 完之后，再执行 execute() 方法。对于模型驱动，只要了解这些就足够了

扩展：模型驱动的底层实现机制

这里用到了 defaultStack 拦截器栈中的 modelDriven 拦截器

它对应 com.opensymphony.xwork2.interceptor.ModelDrivenInterceptor 类，其 API 描述如下

```
public class ModelDrivenInterceptor extends AbstractInterceptor
Watches for ModelDriven actions and adds the action's model on to the
valuestack.
```

翻译：观察模型驱动的动作，并将这个 Action 的模型【这里指 User 对象】放到值栈中

Note: The ModelDrivenInterceptor must come before the both StaticParametersInterceptor and ParametersInterceptor if you want the parameters to be applied to the model.

翻译：若希望将表单提交过来的参数应用到模型里面，那么

ModelDrivenInterceptor 拦截器就必须位于 **StaticParametersInterceptor** 和 **ParametersInterceptor** 拦截器前面。

实际上 `struts-default.xml` 已完成这个工作了。可以在 `defaultStack` 拦截器栈中查看三者位置，所以对于采用模型驱动的方式的话，在 `struts.xml` 中只需要指定模型驱动类就可以了，其它的都不需要我们手工修改

3. 属性驱动接收参数

这种方式应该不算是参数封装的方式，但我们很多情况下都用属性驱动的方式接收参数，因为这种方式方便，简洁，易控制。属性驱动在 **Action** 中提供与表单字段一一对应的属性，然后一一 **set** 赋值，采用属性驱动的方式时，是由每个属性来承载表单的字段值，运转在 **MVC** 流程里面。由于这种方式比较简单，这里就不赘述了。

到底是用属性驱动还是模型驱动呢？

- 1) 统一整个系统中的 **Action** 使用的驱动模型，即要么都是用属性驱动，要么都是用模型驱动。
- 2) 如果你的 **DB** 中的持久层的对象与表单中的属性都是一一对应的，那么就使用模型驱动吧，毕竟看起来代码要整洁得多。
- 3) 如果表单的属性不是一一对应的，那么就应该使用属性驱动，否则，你的系统就必须提供两个 **Bean**，一个对应表单提交的数据，另一个用于持久层。

（四十四）细谈 struts2（七）数据类型转换详解

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

Web 应用程序的交互都是建立在 HTTP 之上的，[互相传递的都是字符串](#)。也就是说服务器接收到的来自用户的数据只能是字符串或者是字符数组，而在 Web 应用的对象中，往往使用了多种不同的类型，如整数(int)、浮点数(float)、日期(Date)或者是自定义数据类型等。因此在服务器端必须将字符串转换成合适的数据类型。

Struts2 框架中为我们提供了一些简单类型的转换器，比如转换为 int、float 等简单数据类型是不需要我们自己定义转换器去转换的，struts2 内部本身就为我们提供了转换的方法，但像一些复杂的类型和我们自定义的数据类型还是需要我们自己去写转换器去转换的。在转换工程中，如果在类型转换中出现异常，类型转换器开发者无需关心异常处理逻辑，Struts2 的 **conversionError** 拦截器会自动处理该异常，并且提示在页面上生成提示信息。

下面我们就一步步的实现和注册一个我们自己的转换器，也就是自定义类型转换器的几个步骤：

一：自定义类型转换器

实现自定义类型转换器我们一般有两种方式：

1.实现 OGNL 提供的 **TypeConvert** 接口以及实现了 **TypeConvert** 接口的 **DefaultTypeConvert** 类来实现自定义的类型转换器。我们来看一下

DefaultTypeConvert 类的源码：

[java] [view plaincopyprint?](#)

```
1. public class DefaultTypeConverter implements TypeConverter{
2.
3.     public DefaultTypeConverter(){
4.
5.         super();
6.
7.     }
8.
9.     /**
10.
11.     * @param context:类型转换的上下文
12.
13.     * @param value:需要转换的参数
14.
15.     * @param toType:转换后的目的类型
16.
17.     */
18.
19.     public Object convertValue(Map context,
20.
21.     Object value,
22.
23.     Class toType)
24.
25.     {
26.
27.         return OgnlOps.convertValue(value, toType);
28.     }
```

```

29. }
30.
31. public Object convertValue(Map context, Object target,
32.
33. Member member, String propertyName,
34.
35. Object value, Class toType)
36.
37. {
38.
39.     return convertValue(context, value, toType);
40.
41. }
42.}

```

convertValue 方法的作用：

该方法负责完成类型的双向转换，为了实现双向转换，我们通过判断 **toType** 的类型即可判断转换的方向。**toType 类型是需要转换的目标类型**，如：当 **toType** 类型是 **User** 类型时，表明需要将字符串转换成 **User** 实例；当 **toType** 类型是 **String** 类型时，表明需要把 **User** 实例转换成字符串类型。通过 **toType** 类型判断了类型转换的方向后，我们就可以分别实现两个方向的转换逻辑了。

实现类型转换器的关键就是实现 **convertValue** 方法，该方法有三个参数：

第一个参数 **context**: 类型转换的上下文

第二个参数 **value**: 需要转换的参数

第三个参数 **toType**: 转换后的目的类型

2. 基于 Struts2 的类型转换器

Struts 2 提供了一个 [StrutsTypeConverter](#) 的抽象类，这个抽象类是 [DefaultTypeConverter](#) 类的子类。开发时可以直接继承这个类来进行转换器的构建。通过继承该类来构建类型转换器，可以不用对转换的类型进行判断(和[第一种方式的区别](#))，下面我们来看一下 [StrutsTypeConverter](#) 类的源码：

[java] [view plain](#)[copy](#)[print?](#)

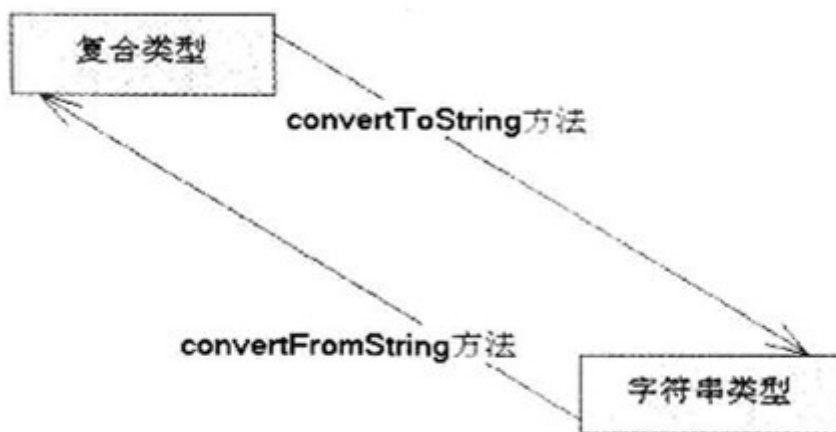
```
1. public abstract class StrutsTypeConverter extends DefaultTypeConverter {
2.
3.     //重写 DefaultTypeConverter 类的 convertValue 方法
4.
5.     public Object convertValue(Map context, Object o, Class toClass) {
6.
7.         //如果需要把复合类型转换成字符串类型
8.
9.         if (toClass.equals(String.class)) {
10.
11.             return convertToString(context, o);
12.
13.         }
14.
15.         //如果需要把字符串转换成符合类型
16.
17.         elseif (o instanceof String[]) {
18.
19.             return convertFromString(context, (String[]) o, toClass);
20.
21.         }
22.
23.         //如果需要把字符串转换成符合类型
24.
25.         elseif (o instanceof String) {
26.
27.             return convertFromString(
```

```

28.
29.context, new String[]{(String) o}, toClass);
30.
31.    } else {
32.
33.        return performFallbackConversion(context, o, toClass);
34.
35.    }
36.
37. }
38.
39.protected Object performFallbackConversion(Map context,
40.
41.Object o, Class toClass) {
42.
43.    return super.convertValue(context, o, toClass);
44.
45. }
46.
47.public abstract Object convertFromString(Map context,
48.
49.String[] values, Class toClass);
50.
51.public abstract String convertToString(Map context, Object o);
52.
53.}

```

该类已经实现了 `DefaultTypeConverter` 的 `convertValue` 方法。实现该方法时，它将两个不同转换方向替换成不同方法——当需要把字符串转换成复合类型时，调用 `convertFromString` 抽象方法；当需要把复合类型转换成字符串时，调用 `convertToString` 抽象方法，下图展示了其对应关系：



二. 注册自定义类型转换器:

实现了自定义的类型转换器之后, 将该类型转换器注册在 Web 应用中, Struts2 框架才可以正常使用该类型转换器, 类型转换器的注册分为两种,

1.注册局部类型转换器。

局部类型转换器仅仅对某个 Action 起作用。局部类型转换器非常简单, 只需要在相应的 Action 目录下新建一个资源文件。该资源文件名格式如下。

ActionName-conversion.properties。其中 ActionName 表示需要进行转换的 Action 的类名, “-conversion.properties”字符串则是固定格式的。该文件也是一个典型 Properties 文件, 文件由键值对组成: **propertyName = 类型转换器类**

如: name=util.NameConvert

name:表示要进行转换的属性

util.NameConvert:表示要进行转换的自定义类型转换器。

注意: 该属性文件应该与 ActionName.class 放在相同位置。

2.注册全局类型转换器。对所有 Action 的特定类型的属性都会生效。

全局类型转换器，必须提供一个 `xwork-conversion.properties` 文件。

文件必须保存在 `classes` 目录下。该资源文件名格式如下：

复合类型=对应的类型转换器

复合类型：指定需要完成类型转换的复合类

对应的类型转换器：指定所指定类型转换的转换器。

如：注册 `User` 类的全局类型转换器为： `UserConverter`

`cn.wjz.bean.User = cn.wjz.util.UserConverter`

注意：如果局部类型转换和全局类型转换同时存在的话，局部类型转换具有较高的优先级，也就是以局部类型转换器为主。

三. 集合类型的类型转换

对于 `List` 元素来说，内容如： `Element_attributeName=typeName`;

对于 `Map` 元素来说，

(1)如果表示 `key` 的类型，则： `Key_attributeName=typeName`;

(2)如果表示 `value` 的类型，则为： `Element_attributeName=typeName`;

比如,此处没有使用泛型，而是使用了局部类型转换文件：

[java] [view plaincopyprint?](#)

```
1. Conversion02Action.java
2.
3. public class Conversion02Action extends ActionSupport {
4.
5.     private List lists;
6.
7.     private Map maps;
8.
```

```
9.    public String execute()throws Exception{
10.
11.        System.out.println(((Person)lists.get(0)).getGender());
12.
13.        System.out.println(((Person)lists.get(0)).getSalary());
14.
15.        System.out.println(((Person)maps.get("one")).getGender());
16.
17.        System.out.println(((Person)maps.get("one")).getSalary());
18.
19.        return SUCCESS;
20.
21.    }
22.
23.    public List getLists() {
24.
25.        return lists;
26.
27.    }
28.
29.    public void setLists(List lists) {
30.
31.        this.lists = lists;
32.
33.    }
34.
35.    public Map getMaps() {
36.
37.        return maps;
38.    }
39.
40.    public void setMaps(Map maps) {
41.
42.        this.maps = maps;
43.
```



```
44.  }  
45.}
```

Conversion02Action-conversion.properties

[html] view plaincopyprint?

1. **Element_lists**= .person.Person
- 2.
3. **Key_maps**= .lang.String
- 4.
5. **Element_maps**= .person.Person

页面表单:

[html] view plaincopyprint?

<s:fielderror></s:fielderror>

<s:form action="conversion02" >

<s:textfield label="list1.salary" name="lists[0].salary"></s:textfield>

<s:textfield label="list1.gender" name="lists[0].gender"></s:textfield>

<s:textfield label="map1.gender" name="maps['one'].gender"></s:textfield>

<s:textfield label="map1.salary" name="maps['one'].salary"></s:textfield>

<s:submit value="提交"></s:submit>

</s:form>

四. **Struts 2** 内建的类型转换器 :

Struts 2 为常用的数据类型提供了内建的类型转换器，所以根本不用自定义转换器。对于内建的转换器，**Struts** 在遇到这些类型时，会自动去调用相应的转换器进行类型转换。

Struts 2 全部的内建转换器:

- 基本数据类型及其封装类**。包括: boolean 和 Boolean、char 和 Character、int 和 Integer、long 和 Integer、float 和 Float、double 和 Double。完成字符串和基本数据类型或其封装类之间的转换。

- 日期类型**。使用当前区域的短格式转换, 即 `DateFormat.getInstance(DateFormat.SHORT)` 完成字符串和日期类型之间的转换。

- 集合 (Collection) 类型**。将 `request.getParameterValues(String arg)` 返回的字符串数据与 `java.util.Collection` 转换。集合元素为 String 类型。

- 集合 (Set) 类型**。与 Collection 的转换相似, 只是去掉了相同的值。集合元素为 String 类型。

- 数组类型**。将 `request.getParameterValues(String arg)` 返回的字符串数组中的每个字符串值取出组成一个数组。数组元素为 String 类型。

注意: Struts 2 提供的全部内建转换器都是**双向**的, 也就是说从用户输入页到服务器端时会将字符串类型转换成相应的数据类型。在显示输出时, 又会将相应的数据类型转换成字符串类型来显

数组类型的转换器。这个转换器非常有用, 比如多个表单元素的 name 属性相同, 那么提交的参数就不再是字符串而是一个字符串数组。通过 Struts 2 提供的数组类型的转换器就能很方便的将多个相同 name 属性的表单元素的值封装到 Action 中的一个数组中。

五. 类型转换中错误处理:

1. 类型转换的错误处理流程:

Struts2 提供了一个名为 `conversionError` 的拦截器，这个拦截器被注册在默认的拦截器栈中，在 `Struts-default.xml` 中的配置信息：

[html] `view plaincopyprint?`

```
1. <interceptor-stack name=                >
2.
3.    ... ..
4.
5.    <!-- 处理类型转换错误的拦截器 -->
6.
7. <interceptor-ref name=                />
8.
9.    <!-- 处理数据校验的拦截器 -->
10.
11.<interceptor-ref name=                >
12.
13.    <param name=                >input,back,cancel,browse</param>
14.
15. </interceptor-ref>
16.
17. <interceptor-ref name=                >
18.
19.    <param name=                >input,back,cancel,browse</param>
20.
21. </interceptor-ref>
22.
23.</interceptor-stack>
```

如果 Struts2 的类型转换器执行类型转换时出现错误，该拦截器将负责将对应的错误封装成表单域错误（**fieldError**），并将这些错误信息放入 `ActionContext` 中。

Struts2 的错误处理流程：

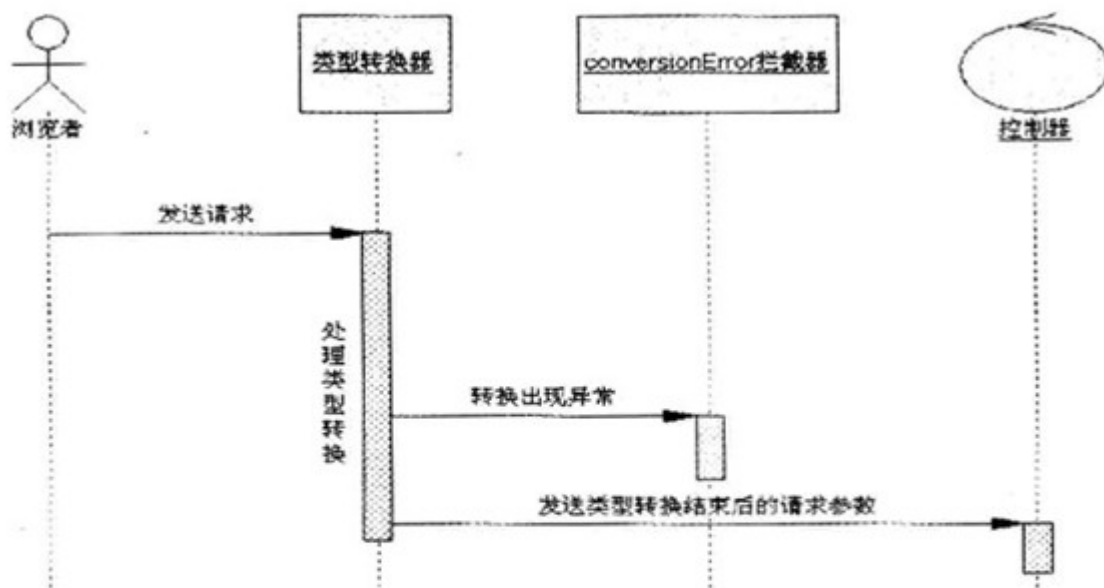


图 5.15 Struts 2 类型转换的错误处理流程

※ 注意 必须指出的是，为了让 Struts 2 框架处理类型转换的错误，以及使用后面的数据校验机制，系统的 Action 类都应该通过继承 ActionSupport 类来实现。ActionSupport 类为完成类型转换错误处理，数据校验实现了许多基础工作。

2、错误信息的友好显示

在进行类型转换中，如果出现错误将会提示错误信息。Struts 2 默认提供了错误信息提示，但是这些错误信息提示不够友好，下面将介绍如何自定义错误信息来取代 Struts 2 的默认错误信息。

·定义全局类型转换错误处理信息：

在应用的国际化资源文件中增加如下的信息：

xwork.default.invalid.fieldvalue = key

key 的值就是用户希望在页面中显示的提示信息。 例如：

#改变默认的类型转换失败后的提示信息

xwork.default.invalid.fieldvalue = {0}字段类型转换失败！！

因为包含非西欧字符，因此使用 **native2ascii** 命令处理

xwork.default.invalid.fieldvalue =

{0}\u5b57\u6bb5\u7c7b\u578b\u8f6c\u6362\u5931\u8d25\u01ff01\u01ff01

·定义局部类型转换错误处理信息：

在某些时候可能还需要对特定的字段指定特别的提示信息，此时可以提供该 Action 的局部资源文件，文件名：**ActionName.properties**，在文件中增加如下一项：

invalid.fieldvalue.属性名 =提示信息

例如：

```
#改变 Action 中 birth 属性类型转换错误后的提示信息
invalid.fieldvalue.birth = 生日信息必须满足 yyyy-MM-DD 格式
使用 native2ascii 命令处理
invalid.fieldvalue.birth =
\u751f\u65e5\u4fe1\u606f\u5fc5\u987b\u6ee1
\u8db3yyyy-MM-DD\u683c\u5f0f
```

六. 类型转换的流程

- 1、用户进行请求，根据请求名在 **struts.xml** 中寻找 Action
- 2、在 Action 中，根据请求域中的名字去寻找对应的 **set** 方法。找到后在赋值之前会检查这个属性有没有自定义的类型转换。没有的话，按照默认进行转换；如果某个属性已经定义好了类型转换，则会去检查在 Action 同一目录下的 **action 文件名-conversion.properties** 文件。
- 3、从文件中找到要转换的属性及其转换类。
- 4、然后进入转换类中，在此类中判断转换的方向。我们是先从用户请求开始的，所以这时先进入从字符串到类的转换。返回转换后的对象。流程返回 Action。

- 5、将返回的对象赋值给 **Action** 中的属性，执行 **Action** 中的 **execute()**
- 6、执行完 **execute()**方法，根据 **struts.xml** 的配置转向页面
- 7、在 **jsp** 中显示内容时，根据页面中的属性名去调用相应的 **get** 方法，以便输出
- 8、在调用 **get** 方法之前，会检查有没有此属性的自定义类型转换。如果有，再次跳转到转换类当中。
- 9、在转换类中再次判断转换方向，进入由类到字符串的转换，完成转换后返回字符串。
- 10、将返回的值直接带出到要展示的页面当中去展示。

（四十五）大话设计模式（九）迭代器模式和命令模式

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

首先来看一下迭代器模式是干什么用的？

迭代这个名词对于熟悉 Java 的人来说绝对不陌生。我们常常使用 JDK 提供的迭代接口进行 java collection 的遍历：

```
Iterator it = list.iterator();
while(it.hasNext()){
    //using "it.next();"do some businesss logic
}
```

而这就是关于迭代器模式应用很好的例子。在软件构建过程中，集合对象内部结构常常变化各异。但对于这些集合对象，我们希望在暴露其内部结构的同时，可以让外部客户代码透明地访问其中包含的元素；同时这种“透明遍历”也为“同一种算法在多种集合对象上进行操作”提供了可能。使用面向对象技术将这种遍历机制抽象为“迭代器对象”为“应对变化中的集合对象”提供了一种优雅的方法。

定义：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。-----《设计模式》GOF

迭代器模式角色组成：

1) 迭代器角色 (Iterator)： 迭代器角色负责定义访问和遍历元素的接口。

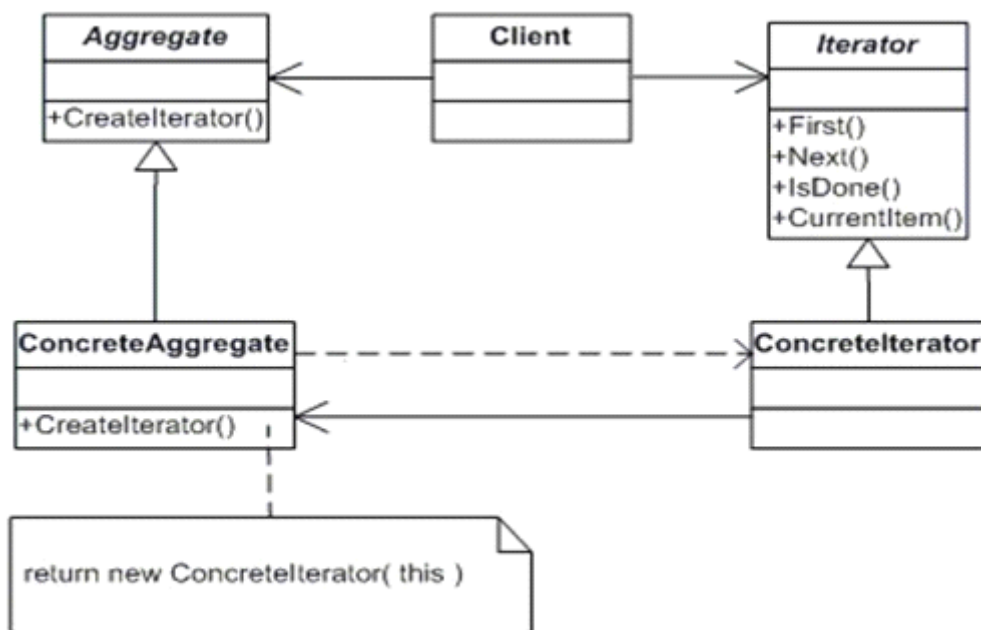
2) 具体迭代器角色 (Concrete Iterator)： 具体迭代器角色要实现迭代器接口，并要记录遍历中的当前位置。

3) 容器角色 (Container) : 容器角色负责提供创建具体迭代器角色的接

口。

4) 具体容器角色 (Concrete Container) : 具体容器角色实现创建具体迭代器角色的接口——这个具体迭代器角色于该容器的结构相关。

结构图 (Struct):



从结构上可以看出，迭代器模式在客户与容器之间加入了迭代器角色。迭代器角色的加入，就可以很好的避免容器内部细节的暴露，而且也使得设计符号“单一职责原则”。

注意，在迭代器模式中，具体迭代器角色和具体容器角色是耦合在一起的——遍历算法是与容器的内部细节紧密相关的。为了使客户程序从与具体迭代器角色耦合的困境中脱离出来，避免具体迭代器角色的更换给客户程序带来的修改，迭代器模式抽象了具体迭代器角色，使得客户程序更具一般性和重用

性。这被称为多态迭代。

适用性：

1. 访问一个聚合对象的内容而无需暴露它的内部表示。
2. 支持对聚合对象的多种遍历。
3. 为遍历不同的聚合结构提供一个统一的接口(即, 支持多态迭代)。

具体的代码如下所示：

Iterator 接口：

[java] [view plaincopyprint?](#)

```
1. package iterator;
2.
3. public interface Iterator{
4.
5.     public Item first();
6.
7.     public Item next();
8.
9.     public boolean isDone();
10.
11.     public Item currentItem();
12.
13. }
```

Controller 类实现了 Iterator 接口。

[java] [view plaincopyprint?](#)

```
1. package iterator;
2.
3. import java.util.Vector;
4.
```

```
5. public class Controller implements Iterator{
6.
7.     private int current = 0;
8.
9.     Vector channel;
10.
11.     public Controller(Vector v){
12.
13.         channel = v;
14.
15.     }
16.
17.     public Item first(){
18.
19.         current = 0;
20.
21.         return (Item)channel.get(current);
22.
23.     }
24.
25.     public Item next(){
26.
27.         current ++;
28.
29.         return (Item)channel.get(current);
30.
31.     }
32.
33.     public Item currentItem(){
34.
35.         return (Item)channel.get(current);
36.
37.     }
38.
39.     public boolean isDone(){
```

```

40.
41.     return current >= channel.size() - 1;
42.
43. }
44.
45.}

```

Television 接口:

[java] [view plain](#)[copy](#)[print?](#)

```

1. package iterator;
2.
3. import java.util.Vector;
4.
5. public interface Television{
6.
7.     public Iterator createIterator();
8.
9.     public Vector getChannel();
10.
11.}

```

HaierTV 类实现了 Television 接口。

[java] [view plain](#)[copy](#)[print?](#)

```

1. package iterator;
2.
3. import java.util.Vector;
4.
5. public class HaierTV implements Television{
6.
7.     private Vector channel;
8.
9.     public HaierTV(){

```

```
10.  
11.     channel =new Vector();  
12.  
13.     channel.addElement(new Item("channel 1"));  
14.  
15.     channel.addElement(new Item("channel 2"));  
16.  
17.     channel.addElement(new Item("channel 3"));  
18.  
19.     channel.addElement(new Item("channel 4"));  
20.  
21.     channel.addElement(new Item("channel 5"));  
22.  
23.     channel.addElement(new Item("channel 6"));  
24.  
25.     channel.addElement(new Item("channel 7"));  
26.  
27. }  
28.  
29. public Vector getChannel(){  
30.  
31.     return channel;  
32.  
33. }  
34.  
35. public Iterator createIterator(){  
36.  
37.     return new Controller(channel);  
38.  
39. }  
40.  
41.}
```

Client 客户端:

[java] [view plaincopyprint?](#)

```

1. package iterator;
2.
3. public class Client{
4.
5.     public static void main(String[] args){
6.
7.         Television tv =new HaierTV();
8.
9.         Iterator it =tv.createIterator();
10.
11.        System.out.println(it.first().getName());
12.
13.        while(!it.isDone()){
14.
15.            System.out.println(it.next().getName());
16.
17.        }
18.
19.    }
20.
21.}

```

Item 类的接口：

[java] [view plaincopyprint?](#)

```

1. package iterator;
2.
3. public class Item{
4.
5.     private Stringname;
6.
7.     public Item(String aName){
8.
9.         name = aName;
10.

```

```
11. }  
12.  
13. public String getName(){  
14.  
15.     return name;  
16.  
17. }  
18.  
19.}  
20.
```

21. 从上面的示例中就可以看出，尽管我们没有显示的引用迭代器，但实质还是通过迭代器来遍历的。总的来说，迭代器模式就是分离了集合对象的迭代行为，抽象出一个迭代器类来负责，这样既可做到不暴露集合的内部结构，又可以让外部代码可以透明的访问集合内部的元素。迭代器模式在访问数组、集合、列表等数据时，尤其是数据库数据操作时，是非常普遍的应用，但由于它太普遍了，所以各种高级语言都对他进行了封装，所以反而给人感觉此模式本身不太常用了。

命令模式

定义：将一个请求封装为一个对象，从而使你不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。别名：动作(Action)、事务(Transaction)

面向对象的程序设计中，一个对象调用另一个对象，一般情况下的调用过程是：创建目标对象实例；设置调用参数；调用目标对象的方法。但在有些情况下有必要使用一个专门的类对这种调用过程加以封装，我们把这种专门的类称作 **command** 类。

命令模式角色组成

1) 命令角色 (Command)：声明执行操作的接口。有 **java** 接口或者抽象类来实现。

2) 具体命令角色 (**Concrete Command**) : 将一个接收者对象绑定于一个动作; 调用接收者相应的操作, 以实现命令角色声明的执行操作的接口。

3) 客户角色 (**Client**) : 创建一个具体命令对象 (并可以设定它的接收者)。

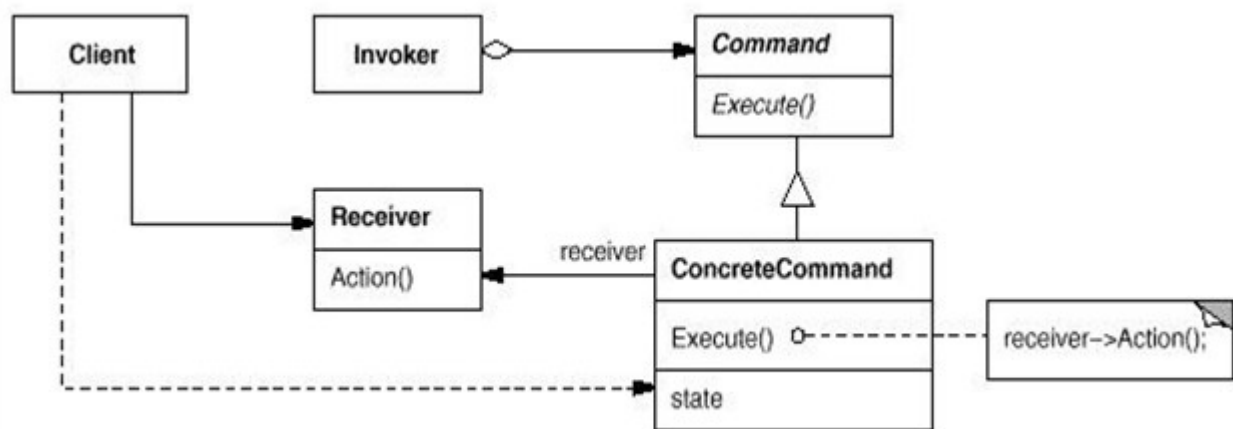
4) 请求者角色 (**Invoker**) : 调用命令对象执行这个请求。

5) 接收者角色 (**Receiver**) : 知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

Command 模式应用范围

- 整个调用过程比较繁杂, 或者存在多处这种调用。这时, 使用 Command 类对该调用加以封装, 便于功能的再利用。
- 调用前后需要对调用参数进行某些处理。
- 调用前后需要进行某些额外处理, 比如日志, 缓存, 记录历史操作等。

结构如下所示:



命令模式的优点:

解耦了发送者和接受者之间联系。发送者调用一个操作,接受者接受请求执行相应的动作,因为使用 **Command** 模式解耦,发送者无需知道接受者任何接口。不少 **Command** 模式的代码都是针对图形界面的,它实际就是菜单命令,我们在一个下拉菜单选择一个命令时,然后会执行一些动作.将这些命令封装成在一个类中,然后用户(调用者)再对这个类进行操作,这就是 **Command** 模式,换句话说,本来用户(调用者)是直接调用这些命令的,如菜单上打开文档(调用者),就直接指向打开文档的代码,使用 **Command** 模式,就是在这两者之间增加一个中间者,将这种直接关系拗断,同时两者之间都隔离,基本没有关系了.显然这样做的好处是符合封装的特性,降低耦合度,**Command** 是将对行为进行封装的典型模式,**Factory** 是将创建进行封装的模式,从 **Command** 模式,我也发现设计模式一个"通病":好象喜欢将简单的问题复杂化,喜欢在不同类中增加第三者,当然这样做有利于代码的健壮性 可维护性 还有复用性.

如何使用?

具体的 **Command** 模式代码各式各样,因为如何封装命令,不同系统,有不同的做法.下面事例是将命令封装在一个 **Collection** 的 **List** 中,任何对象一旦加入 **List** 中,实际上装入了一个封闭的黑盒中,对象的特性消失了,只有取出时,才有可能模糊的分辨出:

典型的 **Command** 模式需要有一个接口.接口中有一个统一的方法,这就是"将命令/请求封装为对象":

```
public interface Command {
```



```
public abstract void execute ( );  
}
```

具体不同命令/请求代码是实现接口 `Command`,下面有三个具体命令

```
public class Engineer implements Command {  
  
    public void execute( ) {  
  
        //do Engineer's command  
  
    }  
}  
public class Programmer implements Command {  
  
    public void execute( ) {  
  
        //do programmer's command  
  
    }  
}  
public class Politician implements Command {  
  
    public void execute( ) {  
  
        //do Politician's command  
  
    }  
}
```

按照通常做法,我们就可以直接调用这三个 `Command`,但是使用 `Command` 模式,我们要将他们封装起来,扔到黑盒子 `List` 里去:

```
public class producer{  
    public static List produceRequests() {  
  
        List queue = new ArrayList();
```

```

        queue.add( new DomesticEngineer() );
        queue.add( new Politician() );
        queue.add( new Programmer() );
        return queue;
    }
}

```

这三个命令进入 **List** 中后,已经失去了其外表特征,以后再取出,也可能无法分辨出谁是 **Engineer** 谁是 **Programmer** 了,看下面客户端如何调用

Command 模式:

```

public class TestCommand {
    public static void main(String[] args) {

        List queue = Producer.produceRequests();
        for (Iterator it = queue.iterator(); it.hasNext(); )

            //客户端直接调用 execute 方法，无需知道被调用者的其它更多类的
            //方法名。

            ((Command)it.next()).execute();

    }
}

```

（四十六）细谈 struts2（八）拦截器的实现原理及源码剖析

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

拦截器（interceptor）是 Struts2 最强大的特性之一，也可以说是 **struts2** 的核心，拦截器可以让你在 Action 和 result 被执行之前或之后进行一些处理。同时，拦截器也可以让你将通用的代码模块化并作为可重用的类。Struts2 中的很多特性都是由拦截器来完成的。拦截是 AOP 的一种实现策略。在 Webwork 的中文文档的解释为：拦截器是动态拦截 Action 调用的对象。它提供了一种机制可以使开发者可以定义在一个 action 执行的前后执行的代码，也可以在一个 action 执行前阻止其执行。同时也是提供了一种可以提取 action 中可重用的部分的方式。谈到拦截器，还有一个词大家应该知道——拦截器链（Interceptor Chain，在 Struts 2 中称为拦截器栈 Interceptor Stack）。拦截器链就是将拦截器按一定的顺序联结成一条链。在访问被拦截的方法或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。

一．拦截器的实现原理：

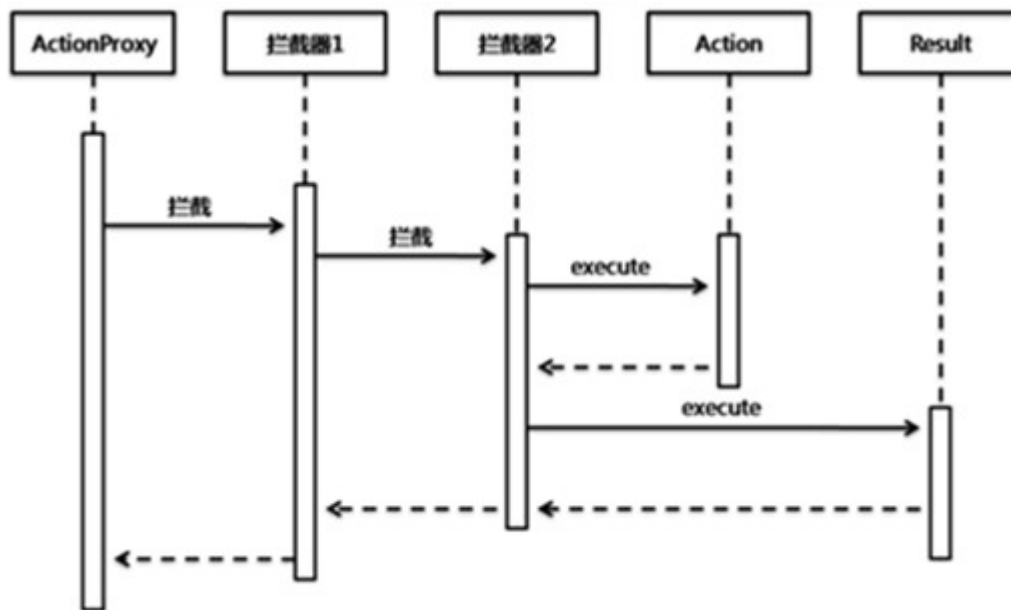
大部分时候，拦截器方法都是通过代理的方式来调用的。**Struts 2** 的拦截器实现相对简单。当请求到达 **Struts 2** 的 **ServletDispatcher** 时，**Struts 2** 会查找配置文件，并根据其配置实例化相对的拦截器对象，然后串成一个列表

（list），最后一个一个地调用列表中的拦截器。事实上，我们之所以能够如此灵活地使用拦截器，完全归功于“动态代理”的使用。动态代理是代理对象

根据客户的需求做出不同的处理。对于客户来说，只要知道一个代理对象就行了。那 Struts2 中，**拦截器是如何通过动态代理被调用的呢**？当 Action 请求到来的时候，会由系统的代理生成一个 Action 的代理对象，由这个代理对象调用 Action 的 execute()或指定的方法，并在 struts.xml 中查找与该 Action 对应的拦截器。如果有对应的拦截器，就在 Action 的方法执行前（后）调用这些拦截器；如果没有对应的拦截器则执行 Action 的方法。其中系统对于拦截器的调用，是通过 ActionInvocation 来实现的。代码如下：

```
if (interceptors.hasNext()) {
    Interceptor interceptor=(Interceptor)interceptors.next();
    resultCode = interceptor.intercept(this);
} else {
    if (proxy.getConfig().getMethodName() == null) {
        resultCode = getAction().execute();
    } else {
        resultCode = invokeAction(getAction(), proxy.getConfig());
    }
}
```

可以发现 Action 并没有与拦截器发生直接关联，而完全是“代理”在组织 Action 与拦截器协同工作。如下图：



二. 拦截器执行分析

我们大家都知道，**Interceptor** 的接口定义没有什么特别的地方，除了 **init** 和 **destory** 方法以外，**intercept** 方法是实现整个拦截器机制的核心方法。而它所依赖的参数 **ActionInvocation** 则是著名的 **Action** 调度者。我们再来看看一个典型的 **Interceptor** 的抽象实现类：

```

public abstract class AroundInterceptor extends AbstractInterceptor {

    /* (non-Javadoc)
     * @see
     * com.opensymphony.xwork2.interceptor.AbstractInterceptor#intercept(com.opensymphony.xwork2.ActionInvocation)
     */

    @Override

```

```

public String intercept(ActionInvocation invocation) throws Exception {

    String result = null;

    before(invocation);

    // 调用下一个拦截器，如果拦截器不存在，则执行 Action

    result = invocation.invoke();

    after(invocation, result);

    return result;

}

public abstract void before(ActionInvocation invocation) throws Exception;

public abstract void after(ActionInvocation invocation, String resultCode)
throws Exception;

}

```

在这个实现类中，实际上已经实现了最简单的拦截器的雏形。这里需要指出的是一个很重要的方法 `invocation.invoke()`。这是 `ActionInvocation` 中的方法，而 `ActionInvocation` 是 `Action` 调度者，所以这个方法具备以下 2 层含义：

1. 如果拦截器堆栈中还有其他的 *Interceptor*，那么 *invocation.invoke()* 将调用堆栈中下一个

Interceptor 的执行。

2. 如果拦截器堆栈中只有 *Action* 了，那么 *invocation.invoke()* 将调用 *Action* 执行。

所以，我们可以发现，*invocation.invoke()* 这个方法其实是整个拦截器框架的实现核心。基于这样的实现机制，我们还可以得到下面 2 个非常重要的推论：

1. 如果在拦截器中，我们不使用 *invocation.invoke()* 来完成堆栈中下一个元素的调用，而是直接返回一个字符串作为执行结果，那么整个执行将被中止。

2. 我们可以以 *invocation.invoke()* 为界，将拦截器中的代码分成 2 个部分，在 *invocation.invoke()* 之前的代码，将会在 *Action* 之前被依次执行，而在 *invocation.invoke()* 之后的代码，将会在 *Action* 之后被逆序执行。

由此，我们就可以通过 *invocation.invoke()* 作为 *Action* 代码真正的拦截点，从而实现 AOP。

三.源码解析

下面我们通过查看源码来看看 Struts2 是如何保证拦截器、*Action* 与 *Result* 三者之间的执行顺序的。之前我曾经提到，*ActionInvocation* 是 Struts2 中的调度器，所以事实上，这些代码的调度执行，是在 *ActionInvocation* 的实现类中完成的，这里，我抽取了 *DefaultActionInvocation* 中的 *invoke()* 方法，它将向我们展示一切。

```

/**

 * @throws ConfigurationException If no result can be found with the returned
code

 */

public String invoke() throws Exception {

    String profileKey = "invoke: ";

    try {

        UtilTimerStack.push(profileKey);

        if (executed) {

            throw new IllegalStateException("Action has already executed");

        }

        // 依次调用拦截器堆栈中的拦截器代码执行

        if (interceptors.hasNext()) {

            final InterceptorMapping interceptor = (InterceptorMapping)
interceptors.next();

            UtilTimerStack.profile("interceptor: "+interceptor.getName(),

            new UtilTimerStack.ProfilingBlock<String>() {

```



```

public String doProfiling() throws Exception {

    // 将 ActionInvocation 作为参数, 调用 interceptor 中
    的 intercept 方法执行

    resultCode =
interceptor.getInterceptor().intercept(DefaultActionInvocation.this);

    return null;

}

});

} else {

resultCode = invokeActionOnly();

}

// this is needed because the result will be executed, then control will
return to the Interceptor, which will

// return above and flow through again

if (!executed) {

    // 执行 PreResultListener

    if (preResultListeners != null) {

        for (Iterator iterator = preResultListeners.iterator();

```

```

iterator.hasNext();) {

    PreResultListener listener = (PreResultListener) iterator.next();

    String _profileKey="preResultListener: ";

    try {

        UtilTimerStack.push(_profileKey);

        listener.beforeResult(this, resultCode);

    }

    finally {

        UtilTimerStack.pop(_profileKey);

    }

}

// now execute the result, if we're supposed to

    // action 与 interceptor 执行完毕，执行 Result

    if (proxy.getExecuteResult()) {

        executeResult();
    }

```

```

    }

    executed = true;

}

return resultCode;

}

finally {

    UtilTimerStack.pop(profileKey);

}

}

```

从源码中，我们可以看到 *Action* 层的 4 个不同的层次，在这个方法中都有体现，他们分别是：拦截器（*Interceptor*）、*Action*、*PreResultListener* 和 *Result*。在这个方法中，保证了这些层次的有序调用和执行。由此我们也可以看出 **Struts2** 在 *Action* 层次设计上的众多考虑，每个层次都具备了高度的扩展性和插入点，使得程序员可以在任何喜欢的层次加入自己的实现机制改变 *Action* 的行为。

在这里，需要特别强调的，是其中拦截器部分的执行调用：

```
resultCode =  
interceptor.getInterceptor().intercept(DefaultActionInvocation.this);
```

表面上，它只是执行了拦截器中的 *intercept* 方法，如果我们结合拦截器来看，就能看出点端倪来：

```
public String intercept(ActionInvocation invocation) throws Exception {  
  
    String result = null;  
  
    before(invocation);  
  
    // 调用 invocation 的 invoke() 方法，在这里形成了递归调用  
  
    result = invocation.invoke();  
  
    after(invocation, result);  
  
    return result;  
  
}
```

原来在 *intercept()* 方法又对 *ActionInvocation* 的 *invoke()* 方法进行递归调用，*ActionInvocation* 循环嵌套在 *intercept()* 中，一直到语句 *result = invocation.invoke()* 执行结束。这样，*Interceptor* 又会按照刚开始执行的逆向顺序依次执行结束。一个有序链表，通过递归调用，变成了一个堆栈执行过程，将一段有序执行的代码变成了 2 段执行顺序完全相反

的代码过程，从而巧妙地实现了 **AOP**。这也就成为了 *Struts2* 的 *Action* 层的 *AOP* 基础。

（四十七）细谈 struts2（九）内置拦截器和自定义拦截器详解(附源码)

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

在上一篇博客中，我们一起看了拦截器的具体实现原理，并且看了一下源码（[细谈 struts2（八）拦截器的实现原理及源码剖析](#)），这一篇博客，我即将带领大家一起来看一下 Struts2 内置实现的拦截器和如何自定义我们自己的拦截器来达到我们想要实现的功能

四. Struts2 内置拦截器

Struts2 中内置类许多的拦截器，它们提供了许多 Struts2 的核心功能和可选的高级特性。这些内置的拦截器在 struts-default.xml 中配置。只有配置了拦截器，拦截器才可以正常的工作和运行。Struts 2 已经为您提供丰富多样的，功能齐全的拦截器实现。大家可以至 struts2 的 jar 包内的 struts-default.xml 查看关于默认的拦截器与拦截器链的配置。内置拦截器虽

然在 **struts2** 中都定义了，但是并不是都起作用的。因为并不是所有拦截器都被加到默认拦截器栈里了，只有被添加到默认拦截器栈里的拦截器才起作用，看一下被加到默认拦截器栈的拦截器都有哪些：

```
<interceptor-stack name="defaultStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="alias"/>
  <interceptor-ref name="servletConfig"/>
  <interceptor-ref name="i18n"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="chain"/>
  <interceptor-ref name="debugging"/>
  <interceptor-ref name="profiling"/>
  <interceptor-ref name="scopedModelDriven"/>
  <interceptor-ref name="modelDriven"/>
  <interceptor-ref name="fileUpload"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="staticParams"/>
  <interceptor-ref name="actionMappingParams"/>
  <interceptor-ref name="params">
    <param name="excludeParams">dojo\..*,^struts\..*</param>
  </interceptor-ref>
  <interceptor-ref name="conversionError"/>
  <interceptor-ref name="validation">
    <param name="excludeMethods">input,back,cancel,browse</param>
  </interceptor-ref>
  <interceptor-ref name="workflow">
    <param name="excludeMethods">input,back,cancel,browse</param>
  </interceptor-ref>
</interceptor-stack>
```

下面我们来学习一下如何在我们的应用中添加其他的拦截器，我们以 **timer** 拦截器为例，**timer** 拦截器可以统计 action 执行的时间。我们可以修改 **package** 中默认的拦截器，那么将替换掉 **struts-default** 中配置的 **defaultStack** 拦截器栈，导致 **Struts2** 无法正常运行，比如无法获取表单的值等等。那么该如何正确的配置呢？可以在添加新的拦截器的基础上加入 **defaultStack** 拦截器栈，这样就可以保证 **defaultStack** 拦截器栈的存在。

[Java] [view plaincopyprint?](#)

```

1. <package name="myStruts" extends="struts-default">
2.
3.   <interceptors>
4.
5.     <interceptor-stack name="myInterceptor"> ①
6.
7.       <interceptor-ref name="timer"/>
8.
9.       <interceptor-ref name="defaultStack"/>
10.
11.     </interceptor-stack>
12.
13.   </interceptors>
14.
15.   <default-interceptor-ref name="myInterceptor"/> ②
16.
17.   <action name="userAction"
18.
19.     class="com.kay.action.UserAction">
20.
21.     <result name="success">suc.jsp</result>
22.
23.     <result name="input">index.jsp</result>
24.
25.     <result name="error">err.jsp</result>
26.
27.   </action>
28.
29.</package>

```

① 添加一个自定义的拦截器栈，并在其中包含 `time` 拦截器和 `defaultStack` 拦截器栈。

② 设置当前的 `package` 的默认拦截器栈为自定义的拦截器栈。

修改 package 的默认拦截器会应用的 package 中的所有 Action 中，如果只想给其中一个 Action 添加拦截器，则可以不修改默认的拦截器栈，只在对应的 Action 添加：

```
<interceptor-ref name="timer"/>
<interceptor-ref name="defaultStack"/>
```

注意，此处一定不要忘记加

<interceptor-ref name="defaultStack"/>，如果忘记的话，struts2 的大部分的功能都实现不了了

五. 定义自己的拦截器

虽然，Struts 2 为我们提供如此丰富的拦截器实现，但是在某种情况下并不能满足我们的需求，比如：访问控制的时候，在用户每次访问某个 action 时，我们要去校验用户是否已经登入，如果没有登入我们将在 action 执行之前就被拦截，此时我们就需要自定义拦截器；下面我们具体看一下，如何实现自定义拦截器。

1.实现拦截器类

所有的 Struts 2 的拦截器都直接或间接实现接口

com.opensymphony.xwork2.interceptor.Interceptor。该接口提供了三个方法：

- 1) void init();在该拦截器被初始化之后，在该拦截器执行拦截之前，系统回调该方法。对于每个拦截器而言，此方法只执行一次。
- 2) void destroy();该方法跟 init()方法对应。在拦截器实例被销毁之前，系统将回调该方法。

3) `String intercept(ActionInvocation invocation) throws Exception`;该方法
是用户需要实现的拦截动作。该方法会返回一个字符串作为逻辑视图。

除此之外，继承类

`com.opensymphony.xwork2.interceptor.AbstractInterceptor` 是更简单的一
种实现拦截器类的方式，因为此类提供了 `init()`和 `destroy()`方法的空实现，这
样我们只需要实现 `intercept` 方法。还有一种实现拦截器的方法是继承
`MethodFilterInterceptor` 类，实现这个类可以实现局部拦截，即可以实现指
定拦截某一个 `action` 的哪个方法，或者不拦截哪个方法

2.注册自定义拦截器

自定义拦截器类实现了，现在就要在 `struts` 里注册这个拦截器；

1) .注册拦截器,在 `struts.xml` 中的 `package` 中注册拦截器

[html] [view plaincopyprint?](#)

```
1. <interceptors>
2.
3.     <!-- name:拦截器的名称, class: 自定义拦截器的类 -->
4.
5.     <interceptorname>interceptorname="拦截器名称"      ="自定义拦截器的
        class 路径"/>
6.
7. </interceptors>
```

2) .使用拦截器,在需要使用自定义拦截器的 `action` 中定义如下代码

[html] [view plaincopyprint?](#)

```
1. <action>
2.
3.     <interceptor-refname>interceptor-refname=           />
4.
```

5. `</action>`

注意：因为 **struts2** 的很多功能都是根据拦截器实现的；如果此处只使用自定义的拦截器时，将失去 **struts2** 的很多核心功能；所以需要定义一个拦截器栈（由一个或多个拦截器组成）

3) 拦截器栈

[html] `view plaincopyprint?`

```
1. <interceptor-stack name=
2.
3.    <!--需要注意的是：系统默认的拦截器栈应要放在前面，在加入自定义拦截器； -->
4.
5.    <interceptor-ref name=
6.
7.    <interceptor-ref name=
8.
9. </interceptor-stack>
```

4) 在 action 中使用栈

[html] `view plaincopyprint?`

```
1. <action>
2.
3.    <interceptor-refnameinterceptor-refname=
4.
5.    . . . . .
6.
7. </action>
```

5) 如果此时需要所有的 **action** 都使用自定义拦截器时，此时就定义一个默认的拦截器

```
<default-interceptor-ref name="permissionStack"/>
```

注意：如果在某个 **action** 中又使用了另一个拦截器，此时默认的拦截器将失效，为了确保能够使用默认的拦截器，又需要添加其他拦截器时，可以在 **action** 中加上其他拦截器

下面咱就以继承 **MethodFilterInterceptor** 类来实现一个权限控制的拦截器，别的页面都不展示了，在此，展示出拦截器类和 **struts.xml** 的配置：

拦截器类 **AuthorInterceptor**:

[java] [view plaincopyprint?](#)

```
1. package com.bzu.inteceptor;
2.
3. import java.util.Map;
4.
5. import javax.servlet.http.HttpServletRequest;
6.
7. import org.apache.struts2.StrutsStatics;
8.
9. import com.opensymphony.xwork2.ActionContext;
10.
11. import com.opensymphony.xwork2.ActionInvocation;
12.
13. import com.opensymphony.xwork2.interceptor.MethodFilterInterceptor;
14.
15.
16. public class AuthorInterceptor extends MethodFilterInterceptor {
17.
```

```
18. @Override
19.
20. protected String doIntercept(ActionInvocation invocation) throws Exception {
21.
22. // TODO Auto-generated method stub
23.
24. ActionContext context = invocation.getInvocationContext();
25.
26.
27. // 通过 ActionContext 来获取 httpRequest
28.
29. HttpServletRequest request = (HttpServletRequest) context
30.
31. .get(StrutsStatics.HTTP_REQUEST);
32.
33. // 也可以通过 ServletActionContext 来获取 httpRequest
34.
35. // HttpServletRequest request = ServletActionContext.getRequest();
36.
37. // 取得根目录的绝对路径
38.
39. String currentURL = request.getRequestURI();
40.
41. // 截取到访问的相对路径，可以通过这个和权限表比较来进行相应的权限控制
42.
43. String targetURL = currentURL.substring(currentURL.indexOf("/", 1),
44.
45. currentURL.length());
46.
47. System.out.println(currentURL + "....." + targetURL);
48.
49.
50. // 通过 ActionContext 获取 session 的信息，以 Map 形式返回
51.
52. Map session = context.getSession();
```

```

53.
54.// 获取容器里面的 username 值，如果存在说明该用户已经登录，让他执行操作，如
    果未登录让他进行登录
55.
56.String username = (String) session.get("username");
57.
58.System.out.println(username+"username");
59.
60.if (username != null) {
61.
62.invocation.invoke();
63.
64.}
65.
66.return "login";
67.}
68.}

```

下面来看一下具体的 `struts.xml` 的配置：

[html] [view plain](#)[copy](#)[print?](#)

```

1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE struts PUBLIC
4.
5.     "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
6.
7.     "http://struts.apache.org/dtds/struts-2.0.dtd">
8.
9.
10.
11.<struts>
12.
13.<constant name="struts.configuration.xml" value="struts.xml" />
14.

```

```

15.
16. <package name=          extends=          >
17.
18.
19.
20. <interceptors>
21.
22.     <!-- 配置未登录进行操作的拦截器 -->
23.
24.     <interceptor name=          class=
25.         >
26.         <param name=          >login</param>
27.
28.     </interceptor>
29.
30.     <!-- 重新封装一个默认的拦截器栈 -->
31.
32.     <interceptor-stack name=          >
33.
34.         <interceptor-ref name=          />
35.
36.         <interceptor-ref name=          />
37.
38.     </interceptor-stack>
39.
40. </interceptors>
41.
42. <!-- 为这个包设置默认的拦截器栈 -->
43.
44. <default-interceptor-ref name=          />
45.
46. <global-results>
47.
48.     <result name=          >/login.jsp</result>

```

```
49.
50. </global-results>
51.
52. <action name=          class=          method=
    >
53.
54. <result name=          >success.jsp</result>
55.
56. <result name=          >fail.jsp</result>
57.
58. <result name=          >login.jsp</result>
59.
60. </action>
61.
62. </package>
63.
64. </struts>
65.
66. </SPAN>
```

以上就是一个简单的权限控制代码实现了。具体的源代码下载地址：[点击下载](#)

最后，大家一起来看一下拦截器与过滤器的区别：

拦截器和过滤器之间有很多相同之处，但是两者之间存在根本的差别。其主要区别为以下几点：

- 1) 拦截器是基于 JAVA 反射机制的，而过滤器是基于函数回调的。
- 2) 过滤器依赖于 Servlet 容器，而拦截器不依赖于 Servlet 容器
- 3) 拦截器只能对 Action 请求起作用，而过滤器可以对几乎所有的请求起作用

用。

4) 拦截器可以访问 Action 上下文、值栈里的对象，而过滤器不能

5) 在 Action 的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次。

（四十八）细谈 struts2（十）ognl 概念和原理详解

引言：众所周知，在 **mvc** 中，数据是在各个层次之间进行流转是一个不争的事实。而这种流转，也就会面临一些困境，这些困境，是由于数据在不同世界中的表现形式不同而造成的：

1. 数据在页面上是一个扁平的，不带数据类型的字符串，无论你的数据结构有多复杂，数据类型有多丰富，到了展示的时候，全都一视同仁的成为字符串在页面上展现出来。

2. 数据在 **Java** 世界中可以表现为丰富的数据结构和数据类型，你可以自行定义你喜欢的类，在类与类之间进行继承、嵌套。我们通常会把这种模型称之为复杂的对象树。

此时，如果数据在页面和 **Java** 世界中互相流转传递，就会显得不匹配。所以也就引出了几个需要解决的问题：

1. 当数据从 **View** 层传递到 **Controller** 层时，我们应该保证一个扁平而分散在各处的数据集合能以一定的规则设置到 **Java** 世界中的对象树中去。同时，能够聪明的进行由字符串类型到 **Java** 中各个类型的转化。

2. 当数据从 **Controller** 层传递到 **View** 层时，我们应该保证在 **View** 层能够以某些简易的规则对对象树进行访问。同时，在一定程度上控制对象树中的数据的数据显示格式。

如果我们稍微深入一些来思考这个问题，我们就会发现，解决数据由于表现形式的不同而发生流转不匹配的问题对我们来说其实并不陌生。同样的问题会发生在 **Java** 世界与数据库世界中，面对这种对象与关系模型的不匹配，我们采用的解决方法是使用 **ORM** 框架，例如 **Hibernate**，**iBatis** 等等。那么现在，在 **Web** 层同样也发生了不匹配，所以我們也需要使用一些工具来帮助我们解决问题。为了解决数据从 **View** 层传递到 **Controller** 层时的不匹配性，**Struts2** 采纳了 **XWork** 的一套完美方案。并且在此的基础上，构建了一个完美的机制，从而比较完美的解决了数据流转中的不匹配性。相信大家看到这一定猜出来了这里的完美方案和完美机制了。对，这就是 **OGNL 方案和 OGNLValueStack 机制**

基本概念

OGNL（**Object Graph Navigation Language**），是一种表达式语言。使用这种表达式语言，你可以通过某种表达式语法，存取 **Java** 对象树中的任意属性、调用 **Java** 对象树的方法、同时能够自动实现必要的类型转化。如果我们把表达式看做是一个带有语义的字符串，那么 **OGNL** 无疑成为了这个语义字符串与 **Java** 对象之间沟通的桥梁。既然 **OGNL** 那么强大，那么让我们一起来研究一下他的 **API**，看看如何使用 **OGNL**。

OGNL 的 **API** 看起来就是两个简单的静态方法：

[java] `view plain`
`copy`
`print?`

1. `public static Object getValue(Object tree, Map context, Object root) throws OgnlException;`
2. `public static void setValue(Object tree, Map context, Object root, Object value) throws OgnlException`

我们可以看到，简单的 **API**，就已经能够完成对各种对象树的读取和设值工作了。这也体现出 **OGNL** 的学习成本非常低。需要特别强调进行区分的，是在针对不同内容进行取值或者设值时，**OGNL** 表达式的不同。**Struts2 Reference** 写道：

The Framework uses a standard naming context to evaluate OGNL expressions. The top level object dealing with OGNL is a Map (usually referred as a context map or context). OGNL has a notion of there being a root (or default) object within the context. In expression, the properties of the root object can be referenced without any special "marker" notion. References to other objects are marked with a pound sign (#).

针对上面的话，我们可以简单的理解为下面两点：

- A) 针对根对象（**Root Object**）的操作，表达式是自根对象到被访问对象的某个链式操作的字符串表示。
- B) 针对上下文环境（**Context**）的操作，表达式是自上下文环境（**Context**）到被访问对象的某个链式操作的字符串表示，但是必须在这个字符串的前面加上#符号，以表示与访问根对象的区别。

OGNL 三要素：

很多人习惯上把传入 **OGNL** 的 **API** 的三个参数，称之为 **OGNL** 的三要素。

OGNL 的操作实际上就是围绕着这三个参数而进行的。

看下面一段测试代码：

[java] [view plaincopyprint?](#)

```
1. Ognl.setValue("department.name", user2, "dev");
2.
3. System.out.println(user2.getDepartment().getName());
4.
5. Ognl.setValue(Ognl.parseexpression_r("department.name"), context, user2, "other Dev");
```

6.

7. `System.out.println(user2.getDepartment().getName());`

1. 表达式 (Expression)

表达式是整个 OGNL 的核心，所有的 OGNL 操作都是针对表达式的解析后进行的。表达式会规定此次 OGNL 操作到底要干什么。我们可以看到，在上面的测试中，`name`、`department.name` 等都是表达式，表示取 `name` 或者 `department` 中的 `name` 的值。OGNL 支持很多类型的表达式，之后我们会看到更多。

2. 根对象 (Root Object)

根对象可以理解为 OGNL 的操作对象。在表达式规定了“干什么”以后，你还需要指定到底“对谁干”。

在上面的测试代码中，`user` 就是根对象。这就意味着，我们需要对 `user` 这个对象去取 `name` 这个属性的值（对 `user` 这个对象去设置其中的 `department` 中的 `name` 属性值）。

3. 上下文环境 (Context)

有了表达式和根对象，我们实际上已经可以使用 OGNL 的基本功能。例如，根据表达式对根对象进行取值或者设值工作。不过实际上，在 OGNL 的内部，所有的操作都会在一个特定的环境中运行，这个环境就是 OGNL 的上下文环境 (Context)。说得再明白一些，就是这个上下文环境 (Context)，将规定 OGNL 的操作“在哪里干”。

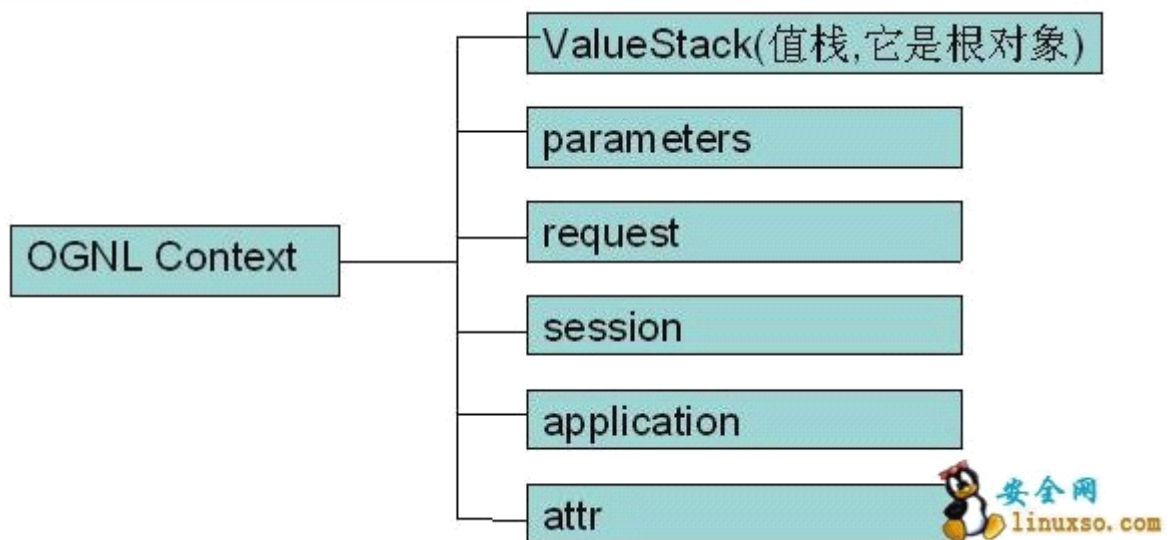
OGNL 的上下文环境是一个 Map 结构，称之为 `OgnlContext`。上面我们提到的根对象 (Root Object)，事实上也会被加入到上下文环境中去，并且这将作为一个特殊的变量进行处理，具体就表现为针对根对象

(Root Object) 的存取操作的表达式是不需要增加 `#` 符号进行区分的。

OgnlContext 不仅提供了 OGNL 的运行环境。在这其中，我们还能设置一些自定义的 parameter 到 Context 中，以便我们在进行 OGNL 操作的时候能够方便的使用这些 parameter。不过正如我们上面反复强调的，我们在访问这些 parameter 时，需要使用#作为前缀才能进行。

OGNL 表达式实现原理

Struts 2 中的 OGNL Context 实现者为 ActionContext, 它结构示意图如下:



当 Struts2 接受一个请求时，会迅速创建 ActionContext, ValueStack, action。然后把 action 存放进 ValueStack，所以 action 的实例变量可以被 OGNL 访问。访问上下文（Context）中的对象需要使用#符号标注命名空间，如#application、#session，另外 OGNL 会设定一个根对象（root 对象），在 Struts2 中根对象就是 ValueStack(值栈)。如果要访问根对象(即 ValueStack)中对象的属性，则可以省略#命名空间，直接访问该对象的属性即可。

在 struts2 中，根对象 ValueStack 的实现类为 OgnlValueStack，该对象不是我们想像的只存放单个值，而是存放一组对象。在 OgnlValueStack 类里有

一个 List 类型的 root 变量，就是使用他存放一组对

象 |--request |--application context -----|--OgnlValueStack root 变量

[action, OgnlUtil, ...] |--session |--attr |--parameters，在 root 变量中处于第一位的对象叫栈顶对象。通常我们在 OGNL 表达式里直接写上属性的名称即可访问 root 变量里对象的属性，搜索顺序是从栈顶对象开始寻找，如果栈顶对象不存在该属性，就会从第二个对象寻找，如果没有找到就从第三个对象寻找，依次往下访问，直到找到为止。 大家注意： Struts2 中，OGNL 表达式需要配合 Struts 标签才可以使用。如：<s:property value="name"/>

由于 ValueStack(值栈)是 Struts 2 中 OGNL 的根对象，如果用户需要访问值栈中的对象，在 JSP 页面可以直接通过下面的 EL 表达式访问 ValueStack(值栈)中对象的属性： \${foo} //获得值栈中某个对象的 foo 属性。如果访问其他 Context 中的对象，由于他们不是根对象，所以在访问时，需要添加#前缀。

application 对象：用于访问 ServletContext，例如#application.userName 或者 #application['userName']，相当于调用 ServletContext 的 getAttribute("username")。

session 对象：用来访问 HttpSession，例如#session.userName 或者 #session['userName']，相当于调用 session.getAttribute("userName")。

request 对象：用来访问 HttpServletRequest 属性（attribute）的 Map，例如 #request.userName 或者 #request['userName']，相当于调用 request.getAttribute("userName")。

parameters 对象：用于访问 HTTP 的请求参数，例如`#parameters.userName`或者`#parameters['userName']`，相当于调用 `request.getParameter("username")`。

attr 对象：用于按 `page->request->session->application` 顺序访问其属性。

好了，基本概念和原理暂时先介绍到这，下篇博客则着重说一下 OGNL 表达式的基本语法和用法，谢谢大家一直以来的支持。

（四十九）细谈 struts2（十一）OGNL 表达式的基本语法和用法

在上篇博客，我们一起看了《ognl 概念和原理详解》，我们大约的知道了 ognl 的基本实现原理和一些基本概念，这节我们一起来学习一下 OGNL 表达式的基本语法和基本用法，首先我们一起来看一下 OGNL 中的#、%和\$符号。

一. OGNL 中的#、%和\$符号

#、%和\$符号在 OGNL 表达式中经常出现，而这三种符号也是开发者不容易掌握和理解的部分。在这里我们简单介绍它们的相应用途。

1. #符号的三种用法

1) 访问非根对象属性，例如示例中的#session.msg 表达式，由于 Struts 2 中值栈被视为根对象，所以访问其他非根对象时，需要加#前缀。实际上，# 相当于 `ActionContext.getContext()`；#session.msg 表达式相当于 `ActionContext.getContext().getSession().getAttribute("msg")`。

2) 用于过滤和投影（projecting）集合，如示例中的 `persons.{?#this.age>20}`。

3) 用来构造 Map，例如示例中的 `#{'foo1':'bar1','foo2':'bar2'}`。

2. %符号

%符号的用途是在标志的属性为字符串类型时，计算 OGNL 表达式的值。如下面的代码所示：

<h3>构造 Map</h3>

```
<s:set name="foobar" value="#{'foo1':'bar1', 'foo2':'bar2'}" />
<p>The value of key "foo1" is <s:property value="#foobar['foo1']" /></p>
<p>不使用%: <s:url value="#foobar['foo1']" /></p>

<p>使用%: <s:url value="%{#foobar['foo1']}" /></p>
```

运行界面如下所示。

he value of key "foo1" is bar1

不使用%: #foobar['foo1']

使用%: bar1

3. \$符号

\$符号主要有两个方面的用途。

1) 在国际化资源文件中，引用 OGNL 表达式，例如国际化资源文件中的代码：reg.agerange=国际化资源信息：年龄必须在\${min}同\${max}之间。

2) 在 Struts 2 框架的配置文件中引用 OGNL 表达式，例如下面的代码片断所示：

[html] view plaincopyprint?

```
1. <validators>
2.
3.   <field name=       >
4.
5.     <field-validator type=       >
6.
7.       <param name=       >10</param>
8.
9.       <param name=       >100</param>
10.
11.     <message>BAction-test 校验：数字必须为${min}为${max}之间！
12.   </message>
```

```
13.     </field-validator>
14.
15.     </field>
16.
17. </validators>
```

二. 我们一起看一下 OGNL 常用表达式:

1. 当使用 OGNL 调用静态方法的时候, 需要按照如下语法编写表达式:

`@package.classname@methodname(parameter)`

2. 对于 OGNL 来说, `java.lang.Math` 是其的默认类, 如果调用 `java.lang.Math` 的静态方法时, 无需指定类的名字, 比如: `@@min(4, 10);`

3. 对于 OGNL 来说, 数组与集合是一样的, 都是通过下标索引来去访问的。

获取 List:`<s:property value="testList"/>
`

获取 List 中的某一个元素(可以使用类似于数组中的下标获取 List 中的内容):

`<s:property value="testList[0]"/>
`

获取 Set:`<s:property value="testSet"/>
`

获取 Set 中的某一个元素(Set 由于没有顺序, 所以不能使用下标获取数据):

`<s:property value="testSet[0]"/>
` ×

获取 Map:`<s:property value="testMap"/>
`

获取 Map 中所有的键:`<s:property value="testMap.keys"/>
`

获取 Map 中所有的值:`<s:property value="testMap.values"/>
`

获取 Map 中的某一个元素(可以使用类似于数组中的下标获取 List 中的内容):

`<s:property value="testMap['m1']"/>
`

获取 List 的大小:`<s:property value="testSet.size"/>
`

4. 使用 OGNL 来处理映射 (Map) 的语法格式如下所示:

`#{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'};`

5. 过滤（filtering）：`collection.{? expression}`
6. OGNL 针对集合提供了一些伪属性（如 `size`，`isEmpty`），让我们可以通过属性的方式来调用方法（本质原因在于集合当中的很多方法并不符合 `JavaBean` 的命名规则），但么你依然还可以通过调用方法来实现与伪属性相同的目的。
7. 过滤（filtering），获取到集合中的第一个元素：`collection.{^ expression}`
8. 过滤（filtering），获取到集合中的最后一个元素：`collection.{& expression}`
9. 在使用过滤操作时，我们通常都会使用 `#this`，该表达式用于代表当前正在迭代的集合中的对象（联想增强的 `for` 循环）
10. 投影（projection）：`collection.{expression}`
11. 过滤与投影之间的差别：类比于数据库中的表，过滤是取行的操作，而投影是取列的操作。具体举例如下：

利用选择获取 List 中成绩及格的对

象：`<s:property value="stus.{?#this.grade>=60}"/>
`

利用选择获取 List 中成绩及格的对象的 username:

`<s:property value="stus.{?#this.grade>=60}.{username}"/>
`

利用选择获取 List 中成绩及格的第一个对象的 username:

`<s:property value="stus.{?#this.grade>=60}.{username}[0]"/>
`

利用选择获取 List 中成绩及格的第一个对象的 username:

`<s:property value="stus.{^#this.grade>=60}.{username}"/>
`

利用选择获取 List 中成绩及格的最后一个对象的 username:

`<s:property value="stus.{ $#this.grade>=60}.{username}"/>
`

利用选择获取 List 中成绩及格的第一个对象然后求大小:

`<s:property value="stus.{^#this.grade>=6000}.{username}.size"/>
`

12. 在 Struts2 中，根对象就是 ValueStack。在 Struts2 的任何流程当中，ValueStack 中的最顶层对象一定是 Action 对象。

13. parameters, #parameters.username

request, #request.username

session, #session.username

application, #application.username

attr, #attr.username

以上几个对象叫做“命名对象”。

14. 访问静态方法或是静态成员变量的改进。

@vs@method

15. 关于 Struts2 标签库属性值的%与#号的关系：

- 1) . 如果标签的属性值是 OGNL 表达式，那么无需加上%{}。
- 2) . 如果标签的属性值是字符串类型，那么在字符串当中凡是出现的%{}都会被解析成 OGNL 表达式，解析完毕后再与其他的字符串进行拼接构造出最后的字符串值。
- 3) . 我们可以在所有的属性值上加%{}，这样如果该属性值是 OGNL 表达式，那么标签处理类就会将%{}忽略掉。

最后一起用代码说话，简单的看一下 ognl 操作的示例：

1) 上下文环境中使用 OGNL

[java] view plaincopyprint?

```
1. public static void main(String[] args)
2.     {
3.         /* 创建一个上下文 Context 对象，它是用保存多个对象一个环境 对象*/
4.         Map<String , Object> context = new HashMap<String , Object>();
5.
6.         Person person1 = new Person();
```

```

7.    person1.setName("zhangsan");
8.
9.    Person person2 = new Person();
10.   person2.setName("lisi");
11.
12.   Person person3 = new Person();
13.   person3.setName("wangwu");
14.
15.   /* person4 不放入到上下文环境中*/
16.   Person person4 = new Person();
17.   person4.setName("zhaoliu");
18.
19.   /* 将 person1、person2、person3 添加到环境中（上下文中）*/
20.   context.put("person1", person1);
21.   context.put("person2", person2);
22.   context.put("person3", person3);
23.
24.   try
25.   {
26.       /* 获取根对象的"name"属性值*/
27.       Object value = Ognl.getValue("name", context, person2);
28.       System.out.println("ognl expression \"name\" evaluation is : " + value);
29.
30.       /* 获取根对象的"name"属性值*/
31.       Object value2 = Ognl.getValue("#person2.name", context, person2);
32.       System.out.println("ognl expression \"#person2.name\" evaluation is : " + value2);
33.
34.       /* 获取 person1 对象的"name"属性值*/
35.       Object value3 = Ognl.getValue("#person1.name", context, person2);
36.       System.out.println("ognl expression \"#person1.name\" evaluation is : " + value3);
37.
38.       /* 将 person4 指定为 root 对象，获取 person4 对象的"name"属性，注意
           person4 对象不在上下文中*/

```

```

39.      Object value4 = Ognl.getValue("name", context, person4);
40.      System.out.println("ognl expression \"name\" evaluation is : " + value4);
41.
42.      /* 将 person4 指定为 root 对象，获取 person4 对象的"name"属性，注意
        person4 对象不在上下文中*/
43.      Object value5 = Ognl.getValue("#person4.name", context, person4);
44.      System.out.println("ognl expression \"person4.name\" evaluation is : " + value5);
45.
46.      /* 获取 person4 对象的"name"属性，注意 person4 对象不在上下文中*/
47.      // Object value6 = Ognl.getValue("#person4.name", context, person2);
48.      // System.out.println("ognl expression \"#person4.name\" evaluation is : " +
        value6);
49.
50.    }

```

2) 使用 **OGNL** 调用方法

[java] [view plaincopyprint?](#)

```

1. public static void main(String[] args)
2.    {
3.        /* OGNL 提供的一个上下文类，它实现了 Map 接口*/
4.        OgnlContext context = new OgnlContext();
5.
6.        People people1 = new People();
7.        people1.setName("zhangsan");
8.
9.        People people2 = new People();
10.       people2.setName("lisi");
11.
12.       People people3 = new People();
13.       people3.setName("wangwu");
14.
15.       context.put("people1", people1);
16.       context.put("people2", people2);

```

```

17.     context.put("people3", people3);
18.
19.     context.setRoot(people1);
20.
21.     try
22.     {
23.         /* 调用 成员方法*/
24.         Object value = Ognl.getValue("name.length()", context, context.getRoot());

25.         System.out.println("people1 name length is :" + value);
26.
27.         Object upperCase = Ognl.getValue("#people2.name.toUpperCase()", context, context.getRoot());
28.         System.out.println("people2 name upperCase is :" + upperCase);
29.
30.         Object invokeWithArgs = Ognl.getValue("name.charAt(5)", context, context.getRoot());
31.         System.out.println("people1 name.charAt(5) is :" + invokeWithArgs);
32.
33.         /* 调用静态方法*/
34.         Object min = Ognl.getValue("@java.lang.Math@min(4,10)", context, context.getRoot());
35.         System.out.println("min(4,10) is :" + min);
36.
37.         /* 调用静态变量*/
38.         Object e = Ognl.getValue("@java.lang.Math@E", context, context.getRoot());
39.         System.out.println("E is :" + e);
40.     }

```

3) 使用 OGNL 操作集合

[java] [view plaincopyprint?](#)

```

1. public static void main(String[] args) throws Exception
2. {

```



```

3.     OgnlContext context = new OgnlContext();
4.
5.     Classroom classroom = new Classroom();
6.     classroom.getStudents().add("zhangsan");
7.     classroom.getStudents().add("lisi");
8.     classroom.getStudents().add("wangwu");
9.     classroom.getStudents().add("zhaoliu");
10.    classroom.getStudents().add("qianqi");
11.
12.    Student student = new Student();
13.    student.getContactWays().put("homeNumber", "110");
14.    student.getContactWays().put("companyNumber", "119");
15.    student.getContactWays().put("mobilePhone", "112");
16.
17.    context.put("classroom", classroom);
18.    context.put("student", student);
19.    context.setRoot(classroom);
20.
21.    /* 获得 classroom 的 students 集合*/
22.    Object collection = Ognl.getValue("students", context, context.getRoot());
23.    System.out.println("students collection is : " + collection);
24.
25.    /* 获得 classroom 的 students 集合*/
26.    Object firstStudent = Ognl.getValue("students[0]", context, context.getRoot());
27.
28.    System.out.println("first student is : " + firstStudent);
29.
30.    /* 调用集合的方法*/
31.    Object size = Ognl.getValue("students.size()", context, context.getRoot());
32.    System.out.println("students collection size is : " + size);
33.
34.    System.out.println("-----飘逸的分割线-----");
35.
36.    Object mapCollection = Ognl.getValue("#student.contactWays", context, context.getRoot());

```

```

36.     System.out.println("mapCollection is :" + mapCollection);
37.
38.     Object firstElement = Ognl.getValue("#student.contactWays[homeNumber]",
context, context.getRoot());
39.     System.out.println("the first element of contactWays is :" + firstElement);
40.
41.     System.out.println("-----飘逸的分割线-----");
42.
43.     /* 创建集合*/
44.     Object createCollection = Ognl.getValue("{aa','bb','cc','dd'", context, context.
getRoot());
45.     System.out.println(createCollection);
46.
47.     /* 创建 Map 集合*/
48.     Object createMapCollection = Ognl.getValue("#{key1':'value1','key2':'value2'}
", context, context.getRoot());
49.     System.out.println(createMapCollection);
50.
51. }
52.}

```

（五十）细谈 Hibernate（一）hibernate 基本概念和体系结构

数据库操作是当今传统应用软件不可缺少的一部分，几乎所用的应用性系统和交互性软件都离不开数据库的支持，所以对数据库数据库的操作也是一个必不可少的工作，在 **java** 的世界里，传统的数据库访问就是 **jdbc** 数据库访问，刚开始学习的时候应该还能满足我们的需求，但真正在实际应用中，其繁琐的操作，开发效率低效，代码冗余等不可避免的缺点也是大家有目共睹的，所以，一套高效简便的数据库访问框架在这种繁琐工作中诞生了，这就是我们如今 **java** 世界里风靡全球的 **Hibernate** 框架（这个应该不夸张吧），所以从今天开始，我和大家就一起进入 **hibernate** 的复习。

Hibernate 百度名片：

Hibernate 是一个开放源代码的**对象**关系映射框架，它对 **JDBC** 进行了非常轻量级的对象封装，使得 **Java** 程序员可以随心所欲的使用对象编程思维来操纵**数据库**。**Hibernate** 可以应用在任何使用 **JDBC** 的场合，既可以在 **Java** 的客户端程序使用，也可以在 **Servlet/JSP** 的 **Web** 应用中使用，最具革命意义的是，**Hibernate** 可以在应用 **EJB** 的 **J2EE** 架构中取代 **CMP**，完成数据持久化的重任。

从上边百度名片中，我们可以看出：

- 1) 其实 **hibernate** 底层依然是 **jdbc** 实现的，只不过 **jdbc** 的繁琐操作都让框架来替我们做了，程序员已经从繁琐的 **jdbc** 操作中解脱出来了。
- 2) **Hibernate** 是一个对象关系映射模型，也就是说，它主要操作的是对象和关系之间的映射，对象，即为我们 **java** 中类的对象，只不过类一般是一些实体类
- 3) **Hibernate** 不仅仅是在 **web** 上的应用框架，这是很多初学者的任务，认为 **hibernate** 只是用在 **web** 开发中的。其实这是一个很错误的观点

持久化：

上面我们看出 **hibernate** 主要完成的是一个数据持久化的重任，很多人应该想了，这个，持久化是什么啊？下面我们就来说一下这个持久化：**持久化是将程序数据在持久状态和瞬时状态间转换的一种机制**，持久化的主要应用是将内存中的对象存储在关系型的数据库中，当然也可以存储在磁盘文件中、**XML** 数据文件中等等。

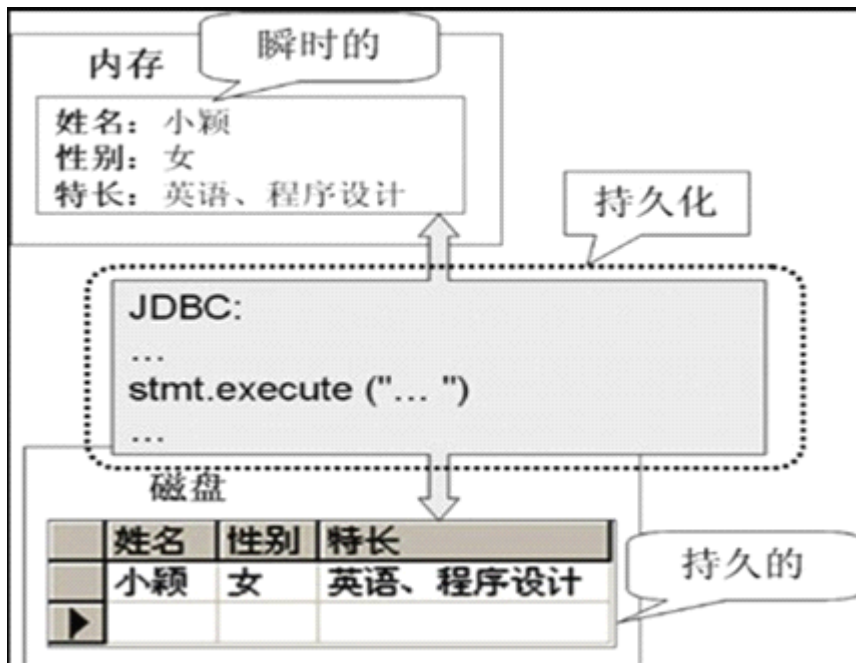
(1) 瞬时状态(transient)

保存在内存中的数据。程序退出后，数据就消失了。

(2) 持久状态(Persisten)

在一定周期内保持不变就是持久化,持久化是针对时间来说的. 数据库中的数据就是持久化了的数据,只要你不删除或修改. 比如在 **IE** 浏览器中一次 **Session** 会话中 **Session** 对象变量也是不变的,是 **Session** 容器中持久化 ,**对象持久化**的方式有很多种,根据周期不同有,**page,Session,Application,**

hibernate 为应用程序提供了高效的 O/R 关系映射和查询服务，为面向对象的领域模型到传统的关系型数据库的映射，提供了一个使用方便的框架。他也是对对象持久化一个很好的实现。简单示例一下：



对象关系映射

从上边我们可以看出，Hibernate 是一个开放源代码的对象关系映射框架，对象/关系数据库映射(object/relational mapping(ORM))这个术语表示一种技术，用来把对象模型表示的对象映射到基于 SQL 的关系模型数据库结构中去。ORM，即 Object- Relational Mapping（对象关系映射），它的作用是在关系型数据库和业务实体对象之间作一个映射，这样，我们在具体的操作业务对象的时候，就不需要再去和复杂的 SQL 语句打交道，只要像平时操作对象一样操作它就可以了。对象关系映射（ORM）提供了概念性的、易于理解的模型化数据的方法。ORM 方法论应当基于三个核心原则：

简单：以最基本的形式建模数据。

传达性：数据库结构被任何人都能理解的语言文档化。

精确性：基于数据模型创建正确标准化了的结构。

下面我们就一起来认识一下 hibernate，**Hibernate** 能做什么？

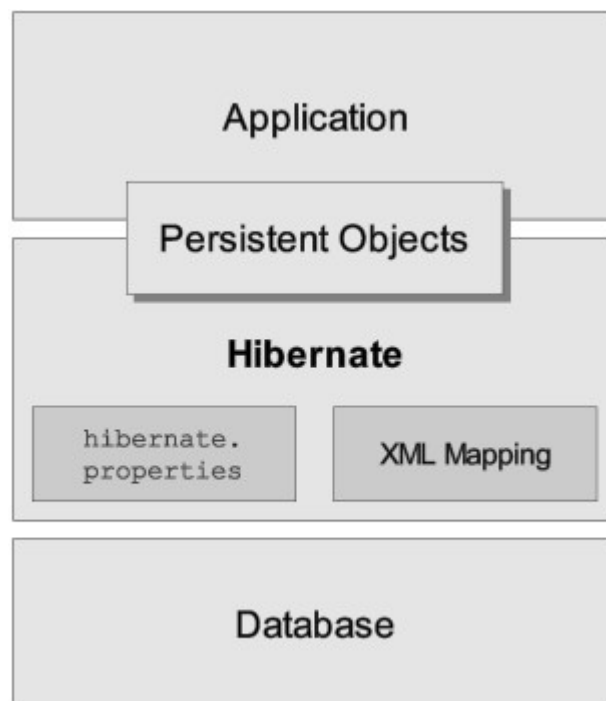
Hibernate 能帮助我们利用面向对象的思想，开发基于关系型数据库的应用程序

第一：将对象数据保存到数据库

第二：将数据库数据读入对象中

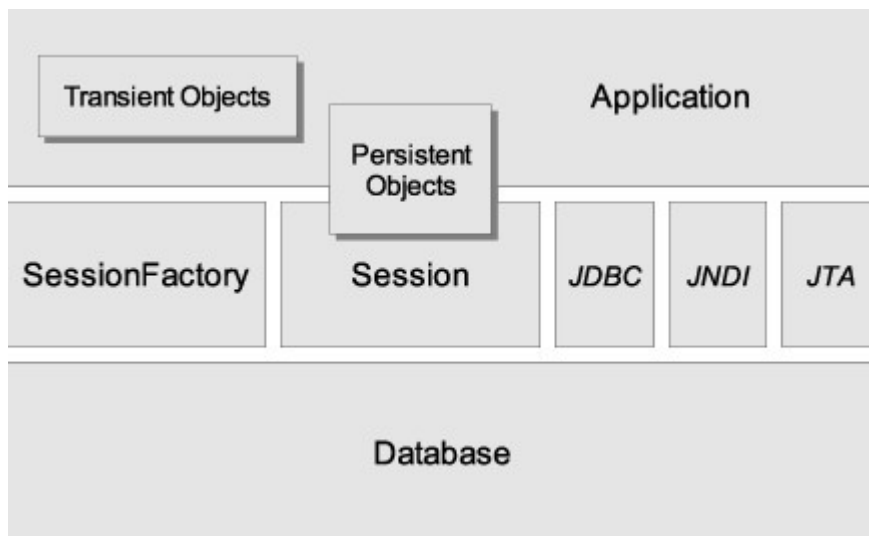
Hibernate 体系结构：

一个非常简要的 **Hibernate** 体系结构的概要图：



从这个图可以看出，**Hibernate** 使用数据库和配置信息来为应用程序提供持久化服务（以及持久的对象）。

我们来更详细地看一下 **Hibernate** 运行时体系结构。由于 **Hibernate** 非常灵活，且支持多种应用方案，所以我们只描述一下两种极端的情况。“轻型”的体系结构方案，要求应用程序提供自己的 **JDBC** 连接并管理自己的事务。这种方案使用了 **Hibernate API** 的最小子集：



“全面解决”的体系结构方案，将应用层从底层的 **JDBC/JTA API** 中抽象出来，而让 **Hibernate** 来处理这些细节。

好了，**hibernate** 基本概念和体系结构就先介绍到这，下一篇博客我将会和大家一起来开发我们的第一个 **hibernate** 应用程序，谢谢大家支持。

（五十一）细谈 Hibernate（二）开发第一个 hibernate 基本详解

在上篇博客中，我们介绍了《[hibernate 基本概念和体系结构](#)》，也对 hibernate 框架有了一个初步的了解，本文我将向大家简单介绍 **Hibernate** 的核心 API 调用库，并讲解一下它的基本配置。核心 API 的底层实现和源码解析将在以后的博客中一一为大家讲解。

首先我们一起来看一下开发一个 hibernate 应用程序的大体流程是什么样的（流程顺序可以颠倒）：

- 创建 **Hibernate** 的配置文件
- 创建持久化类
- 创建对象-关系映射文件
- 通过 **Hibernate API** 编写访问数据库的代码

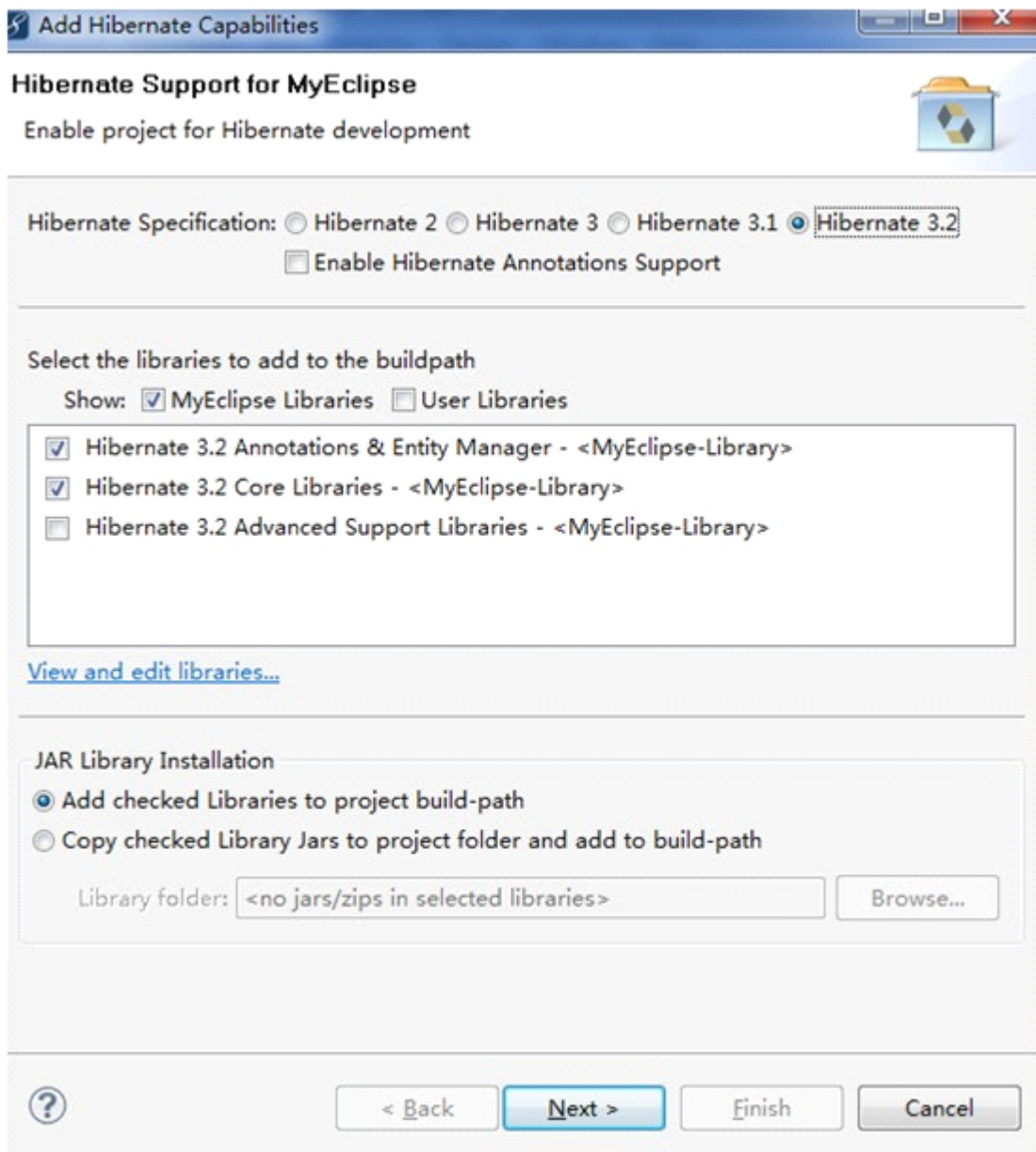
关于配置文件推荐阅读：[细谈 Hibernate（四）Hibernate 常用配置文件详解](#)

知道了开发流程，那么我们就开始我们的第一 hibernate 应用程序，首先我们以一个简单的学生管理应用程序来作为我们第一 hibernate 应用程序的开发，让大家熟悉一下 hibernate 的开发流程，以及对相对应功能的 API 有初步的认识，**Hibernate** 应用程序定义了一些持久类，并且定义了这些类与数据库表格的映射关系。在我们这个简单的学生管理应用程序中包含了一个类和一个映射文件。让我们看看这个简单的持久类包含有一些什么？映射文件是怎样定义的？另外，我们该怎样用 **Hibernate** 来操作这个持久类。

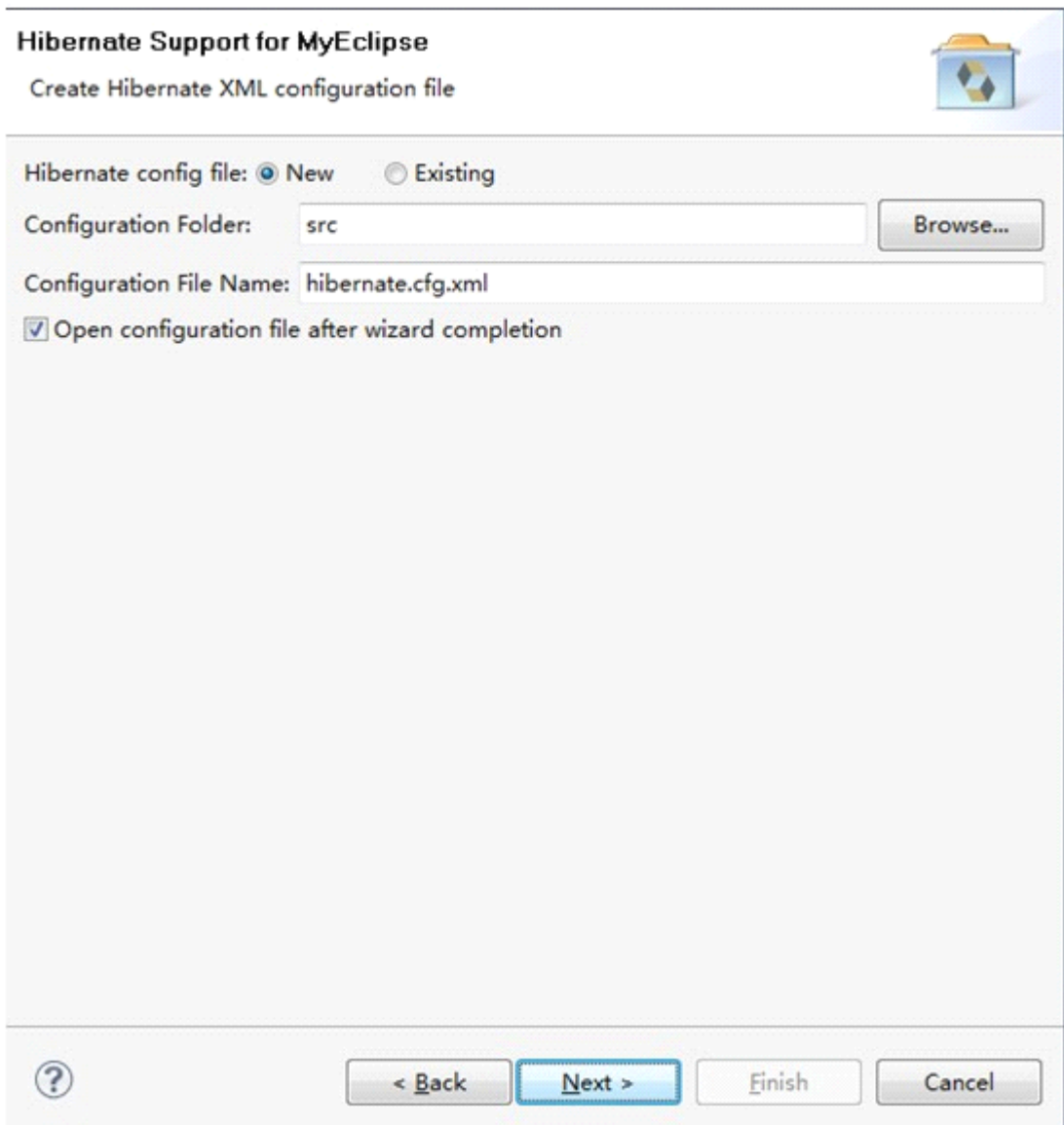
Hibernate 开发全面流程和开发配置

首先创建一个项目。然后创建配置开发环境，主要步骤如下：

MyEclipse-->ProjectCapabilities-->add Hibernate Capabilities,视图如下：



复选框选中第二个 copychecke。。。。那一个。然后点击 next;



为了锻炼能力,配置文件第一次手写,不用图形界面自动生成,所以把: Open configurationfile after wizard completion 选中的点掉; 点击 next

☐ Specify database connection details?

DataSource: ☒ Use JDBC Driver ☐ Use JNDI DataSource

DB Driver:

☒ Copy DB driver jar(s) to project and add to buildpath?

Connect URL:

Driver Class:

Username:

Password:

Dialect:

☐ Enable dynamic DB table creation

Specify database connection details 把这也点掉； 点击 next;

☒ Create SessionFactory class?

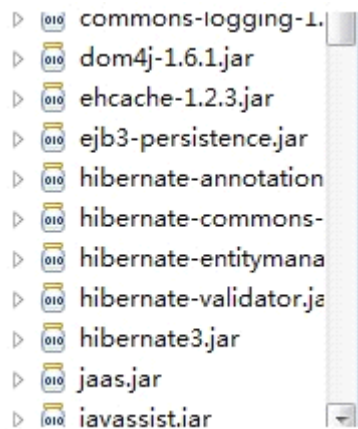
Java source folder:

Java package:

Class name:

Java Compliance Level: ☐ Java 1.4 ☒ Java 5

全部都用手写，所以把 `create sessionFactory class` 此处也点掉；点击 `Finish`；
然后 `lib` 里面会自动增加一系列 `jar` 包



并且 `src` 里面也生成了 `hibernate.cfg.xml`，：

由于本人此次练习的是整合 `Struts` 与 `Hibernate` 进行应用的开发

所以还要进行 `struts` 相关文件的配置，由于 `struts` 相关文件配置比较简单，
所以此处省略；

首先编写一个持久化类：本人创建的是 `person` 类：模型类即为 `javabean`，
很简单；

注：持久化类符合 **JavaBean** 的规范，包含一些属性，以及与之对应的
`getXXX()`和 **`setXXX()`**方法。

- 持久化类有一个 **`id`** 属性，用来惟一标识 **`Person`** 类的每个对象。在面向对象术语中，这个 **`id`** 属性被称为对象标识符（**`OID`**，**`Object Identifier`**）

- Hibernate** 要求持久化类必须提供一个不带参数的默认构造方法

下一步即写一个提交到数据库内容的表单，简单的表单不在此处写了。然后
写一个 `action`，获得表单的内容：此处 `action` 中的处理方法：**`execute ()`**
里面主要是把表单所要提交的信息调用模型类封装成一个类

下一步即将和 **hibernate** 的 **API** 打交道了。。。编写工具类 **HibernateUtil** 这个类主要是获得和数据库打交道的 **Session** 类

[java] [view plaincopyprint?](#)

```
1. 主要代码:
2.
3. private static SessionFactory sessionFactory;
4.
5. //由于一共只有一个 SessionFactory 所以把创建 SessionFactory 的代码放到 static
   代码块里面, 让他只创建一次
6.
7.     static {
8.
9.         try {
10.
11.             /**
12.
13.             * 创建 SessionFactory 过程:
14.
15.             * 1.new Configuration().configure()获得一个 configuration 的实例
16.
17.             * 2.configuration.buildSessionFactory()获得创建 session 的工厂
               sessionFactory 的实例
18.
19.注: Configuration 类用来管理我们的配置文件的信息的, 通过它, 我们可以通过创
   建一个 configuration 实例来管理相应的配置文档, 但是通常我们只创建一个
   configuration 实例。
20.
21.         */
22.
23.         sessionFactory = new Configuration().configure()
24.
25.             .buildSessionFactory();
26.
```

```

27.     } catch (Exception e) {
28.
29.         e.printStackTrace();
30.
31.     }
32.
33. }
34.
35. publicstatic Session openSession(){
36.
37.     //获得与数据库打交道的实例 session
38.
39.     Session session=sessionFactory.openSession();
40.
41.     //把工具 session 返回给调用者
42.
43.     return session;
44.
45. }

```

注意：此 Session 非彼 Session(HttpSession)

一个 SessionFactory 实例对应一个数据存储源，应用从 SessionFactory 中获得 Session 实例。

SessionFactory 有以下特点：

- 它是线程安全的，这意味着它的同一个实例可以被应用的多个线程共享。
- 它是重量级的，这意味着不能随意创建或销毁它的实例。如果应用只访问一个数据库，只需要创建一个 SessionFactory 实例，在应用初始化的时候创建该实例。如果应用同时访问多个数据库，则需要为每个数据库创建一个单独的 SessionFactory 实例。

Session 接口是 **Hibernate** 应用使用最广泛的接口。

- Session** 也被称为持久化管理器，它提供了和持久化相关的操作，如添加、更新、删除、加载和查询对象。

- Session** 有以下特点：

- 不是线程安全的，因此在设计软件架构时，应该避免多个线程共享同一个 **Session** 实例。

- Session** 实例是轻量级的，所谓轻量级是指它的创建和销毁不需要消耗太多的资源。这意味着在程序中可以经常创建或销毁 **Session** 对象，例如为每个客户请求分配单独的 **Session** 实例，或者为每个工作单元分配单独的 **Session** 实例。

Session 接口提供了操纵数据库的各种方法，如：

- save()**方法：把 **Java** 对象保存数据库中。

- update()**方法：更新数据库中的 **Java** 对象。

- delete()**方法：把 **Java** 对象从数据库中删除。

- get()**方法：从数据库中加载 **Java** 对象。

然后在编写用 **session** 处理数据类 **DAO** 类，先编写有关接口然后编写实现类 **personDaoImpl**，这个类主要用 **session** 处理数据执行事务的类：

示例代码：

[java] [view plaincopyprint?](#)

1. //调用工具类获得 **session**
- 2.
3. **Session session = factory.openSession();**

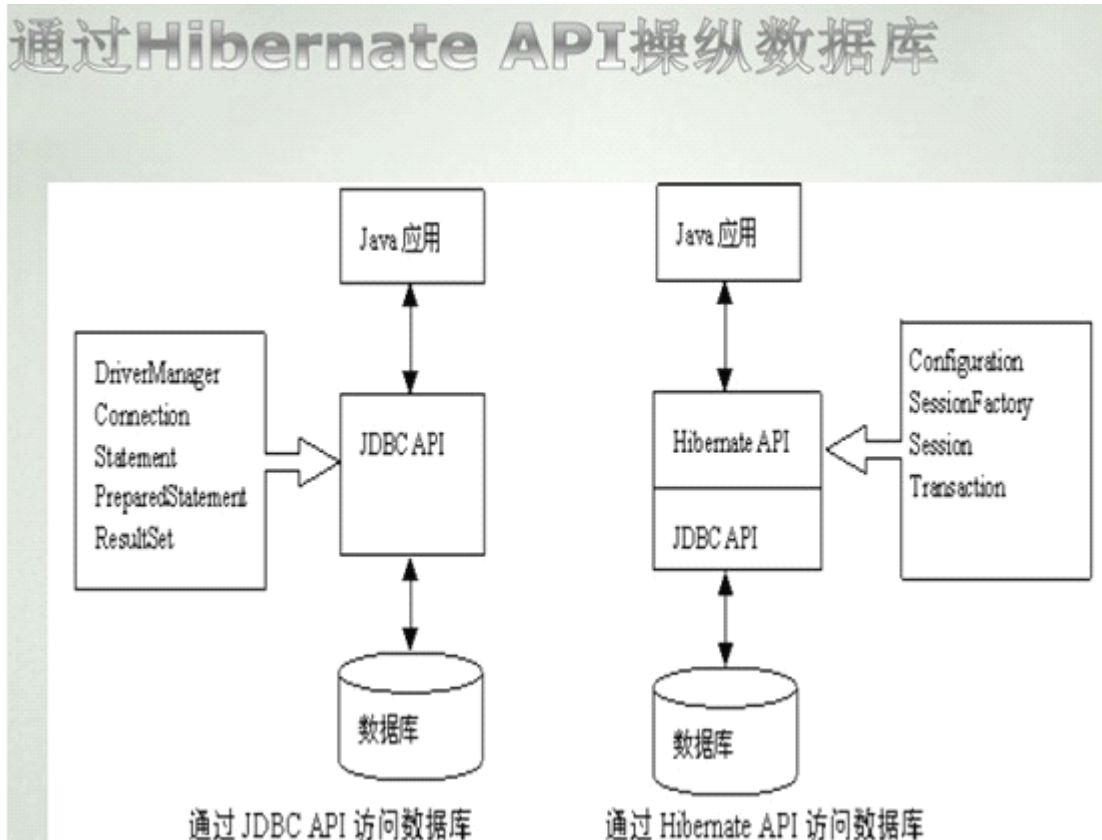
```
4.
5. Transaction tx;
6.
7. try {
8.
9. //开始一个事务
10.
11.tx = session.beginTransaction();
12.
13.//执行事务
14.
15.Student stu=new Student("李华","男","山东");
16.
17.
18.
19.Session.save(stu);
20.
21.//提交事务
22.
23.tx.commit();
24.
25.}
26.
27.catch (Exception e) {
28.
29.//如果出现异常，就撤销事务
30.
31.if (tx!=null) tx.rollback();
32.
33.throwe;
34.
35.}
36.
37.finally {
38.
```



```

39.//不管事务执行成功与否，最后都关闭 Session
40.
41.session.close();
42.
43.}

```



然后编写逻辑处理类（服务类）：先写有关接口,然后写实现类.这个类主要用于处理一些逻辑处理，在这个地方主要用于调用数据处理类 **DAO** 进行数据处理

[java] [view plaincopyprint?](#)

```

1. publicclass PersonServiceImpl implements PersonService {
2.
3.
4.

```

```

5.    public void savePerson(Person person)
6.
7.    {
8.
9.        PersonDAO personDAO = new PersonDAOImpl();
10.
11.        personDAO.savePerson(person);
12.
13.    }
14.
15.}

```

然后直接在 **action** 里面写出 **PersonServiceImpl** 实例进行调用其方法进行就可以了。**Action** 里面主要处理方法代码示例

[java] [view plain](#) [copy print?](#)

```

1. <SPAN xmlns="http://www.w3.org/1999/xhtml">public String execute() throws Exc
   eption
2.
3.    {
4.
5.        Person person = new Person();
6.
7.
8.
9.        person.setUsername(username);
10.
11.        person.setPassword(password);
12.
13.        person.setAge(age);
14.
15.        java.sql.Date registerDate = new java.sql.Date(new java.util.Date().g
        etTime());

```

```

16.
17.     System.out.println("222222");
18.
19.     person.setRegisterDate(registerDate);
20.
21.
22.
23.     PersonServiceImpl personService = new PersonServiceImpl();
24.
25.
26.
27.     personService.savePerson(person);
28.
29.
30.
31.     return SUCCESS;
32.
33.
34.
35. }
36. </SPAN>

```

写到这，一个 **hibernate** 主要的逻辑层就差不多了，现在就主要看怎样写配置文件了：先配置 **hibernate.cfg.xml**.主要配置示例：

[html] [view plaincopyprint?](#)

```

1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE hibernate-configuration PUBLIC
4.
5.     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
6.
7.     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

```

```

8.
9.
10.
11. <hibernate-configuration>
12.
13.
14.
15. <session-factory>
16.
17. <property name=                >jdbc:sqlserver://localhost:1433; DatabaseNa
    me=                </property>
18.
19. <property name=                >sa</property>
20.
21. <property name=                >123456</property>
22.
23. <property name=                >com.microsoft.sqlserver.jdbc.SQ
    LServerDriver</property>
24.
25. <property name=                >org.hibernate.dialect.SQLServerDialect</property>
26.
27. <property name=                >true</property>
28.
29. <mapping resource=                >//映射到具体 person 表的配置文件
30.
31. </session-factory>
32.
33.
34.
35. </hibernate-configuration></SPAN>

```

注意：

Hibernate 的描述文件中存放数据库连接驱动程序类，登陆数据库的用户名/密码，映射实体类配置文件的位置等信息。

•将该配置文件存放在 **src** 目录下

描述文件相关属性描述含义：

Hibernate配置文件的属性	
属性	描述
hibernate.dialect	指定数据库使用的 SQL 方言
hibernate.connection.driver_class	指定数据库的驱动程序
hibernate.connection.url	指定连接数据库的 URL
hibernate.connection.username	指定连接数据库的用户名
hibernate.connection.password	指定连接数据库的口令
hibernate.show_sql	如果为 true，表示在程序运行时，会在控制台输出 SQL 语句，这有利于跟踪 Hibernate 的运行状态。默认为 false。在应用开发和测试阶段，可以把这个属性设为 true，以便跟踪和调试应用程序，在应用发布阶段，应该把这个属性设为 false，以便减少应用的输出信息，提高运行性能。

最后一步：

对持久化（实体）类 **Person.java** 文件创建一个 **Hibernate** 映射文件

Person.hbm.xml

Java 的实体类是通过配置文件与数据表中的字段相关联。**Hibernate** 在运行时解析配置文件，根据其中的字段名生成相应的 **SQL** 语句

•将该文件存放在 **src** 目录下

具体代码示例：

[html] view plaincopyprint?

```
1. <SPAN xmlns=
2.
3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.
   0//EN"
4.
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6.
7.
8.
9. <hibernate-mapping>
10.
11. <class name=
    table=
    >
12.
13. <id name=
    column=
    type=
    >
14.
15. <generator class=
    ></generator>
16.
17. </id>
18.
19. <property name=
    column=
    type=
    ></property>
20.
21. <property name=
    column=
    type=
    ></property>
22.
23. <property name=
    column=
    type=
    ></property>
24.
25. <property name=
    column=
    type=
    ></propert
    y>
26.
27. </class>
28.
29. </hibernate-mapping></SPAN>
```

注：id 元素对应的一定是数据库的主键列；class="increment"意为自增

- <generator>子元素用来设定标识符生成器。Hibernate提供了多种内置的实现。

标识符生成器	描述
increment	适用于代理主键。由 Hibernate 自动以递增的方式生成标识符，每次增量为 1。
identity	适用于代理主键。由底层数据库生成标识符。前提条件是底层数据库支持自动增长字段类型。
sequence	适用于代理主键。Hibernate 根据底层数据库的序列来生成标识符。前提条件是底层数据库支持序列。
hilo	适用于代理主键。Hibernate 根据 high/low 算法来生成标识符。Hibernate 把特定表的字段作为 “high” 值。默认情况下选用 hibernate_unique_key 表的 next_hi 字段。
native	适用于代理主键。根据底层数据库对自动生成标识符的支持能力，来选择 identity、sequence 或 hilo。
uuid.hex	适用于代理主键。Hibernate 采用 128 位的 UUID (Universal Unique Identification) 算法来生成标识符。UUID 算法能够在网络环境中生成惟一的字符串标识符。这种标识符生成策略并不流行，因为字符串类型的主键比整数类型的主键占用更多的数据库空间。
assigned	适用于自然主键。由 Java 应用程序负责生成标识符，为了能让 Java 应用程序设置 OID，不能把 setId() 方法声明为 private 类型。应该尽量避免使用自然主键。

<property>元素映射值 类型属性

- name 属性：指定持久化类的属性的名字。
- type 属性：指定 Hibernate 或 Java 映射类型。Hibernate 映射类型是 Java 类型与 SQL 类型的桥梁。
- column 属性：指定与类的属性映射的表的字段名。

Java类型、Hibernate映射类型以及SQL类型之间的对应关系

Java类型	Hibernate类型	Sql类型
java.lang.String	string	Varchar
int	int	int
char	character	char(1)
boolean	boolean	bit
java.lang.String	text	text
byte[]	binary	blob
java.sql.Date	date	date
java.sql.Timestamp	timestamp	timestamp

至此，hibernate 开发过程和相关配置就此完毕

（五十二）细谈 Hibernate （三）Hibernate 常用 API 详解及源码分析

新接触一个框架的目的就是想利用这个框架来为我们做一些工作，或者是让他来简化我们的工作，利用这个框架无非就是要利用这个框架所给我们提供的 API 去操作我们的数据，所以利用一个框架的好坏很大一部分取决于你对这个框架 API 的理解程度，所以在此篇博客中我们就一起来看一下

Hibernate 的 API 和配置文件的相信情况。下面我们一一起来看一下：

一：使用 SchemaExport 自动创建数据库表

我相信在此之前大家应该都是用最原始的方法：SQL 建立数据库相关的表，然后再 Java 写映射写配置文件.现在我们学习了 hibernate 以后就可以用一些偷懒的方式了，我们可以使用 SchemaExport 自动创建数据库，建立根据你的对象建立数据库表。下面我们来看一下具体操作：

首先当然要建好 POJO object, XML Mapping File(也可以使用工具根据 POJO class 建立)，配置文件(hibernate.cfg.xml)，然后运行下面的 Java 代码

[java] [view plaincopyprint?](#)

```
1. import org.hibernate.cfg.Configuration;
2.
3. import org.hibernate.tool.hbm2ddl.SchemaExport;
4.
5.
6. public class SchemaUtil {
7.     public static void main(String[] args) {
8.
9.         Configuration cfg = new Configuration().configure();
10.
```

```
11.     SchemaExport schemaExport= new SchemaExport(cfg);
12.
13.     schemaExport.create(false, true);
14.
15. }
16.}
```

再看看数据库，表是不是已经帮你建好了，对于我这样不熟悉数据库的人真是太方便了。

二. 使用 **Hibernate** 操作数据库需要七个步骤

(1) 读取并解析配置文件

```
Configuration conf = new Configuration().configure();
```

(2) 读取并解析映射信息，创建 **SessionFactory**

```
SessionFactory sf = conf.buildSessionFactory();
```

(3) 打开 **Session**

```
Session session = sf.openSession();
```

(4) 开始一个事务（增删改操作必须，查询操作可选）

```
Transaction tx = session.beginTransaction();
```

(5) 数据库操作

```
session.save(user);//或其它操作
```

(6) 提交事务（回滚事务）

```
tx.commit();(tx.rollback();)
```

(7) 关闭 **session**

```
session.close();
```

下面我们来详细看一下这七大步骤的 API:

Configuration:负责管理 Hibernate 的配置信息，这些配置信息都是从配置文件 hibernate.cfg.xml 或者 Hibernate.properties 读取的，当然也可以自定义文件

名称，只要在实例化 Configuration 的时候指定具体的路径就可以了；他为什么会自动加载 hibernate.cfg.xml 文件的呢？我们看一下 configure 源码就一目了然了

[Java] [view plaincopyprint?](#)

```
1. public Configuration configure() throws HibernateException {  
2.  
3.     configure( "/hibernate.cfg.xml" );  
4.  
5.     return this;  
6.  
7. }
```

从这里我们可以看出，在 hibernate 源码中，他就是默认的加载 hibernate.cfg.xml，当然你也可以指定加载配置文件，**Configuration** 提供了相应的方法：

```
public Configuration configure(String resource)  
public Configuration configure(URL url)  
public Configuration configure(File configFile)
```

SessionFactory：Configuration 的实例会根据当前的配置信息，构造

SessionFactory 实例。SessionFactory 是线程安全的，一般情况下一个应用中的一个数据库共享一个 SessionFactory 实例。

Hibernate 的 SessionFactory 接口提供 Session 类的实例，Session 类用于完成对数据库的操作。由于 SessionFactory 实例是线程安全的（而 Session 实例不是线程安全的），所以每个操作都可以共用同一个 SessionFactory 来获取 Session。

Hibernate 配置文件分为两种格式，一种是 xml 格式的配置文件，另一种是 Java 属性文件格式的配置文件，因此构建 SessionFactory 也有两种方法，下面分别介绍。

1. 从 XML 文件读取配置信息构建 SessionFactory

从 XML 文件读取配置信息构建 SessionFactory 的具体步骤如下。

(1) 创建一个 Configuration 对象，并通过该对象的 configure()方法加载 Hibernate 配置文件，代码如下。

```
Configuration config = new Configuration().configure();
```

(2) 完成配置文件和映射文件的加载后，将得到一个包括所有 Hibernate 运行期参数的 Configuration 实例，通过 Configuration 实例的 buildSessionFactory()方法可以构建一个惟一的 SessionFactory，代码如下。

```
SessionFactory sessionFactory = config.buildSessionFactory();
```

构建 SessionFactory 要放在静态代码块中，因为它只在该类被加载时执行一次。

2 从 Java 属性文件读取配置信息构建 SessionFactory

从 Java 属性文件读取配置信息构建 SessionFactory 的具体步骤如下。

(1) 创建一个 Configuration 对象，此时 Hibernate 会默认加载 classpath 中的配置文件 hibernate.properties，代码如下。

```
Configuration config = new Configuration();
```

(2) 由于在配置文件中缺少相应的配置映射文件的信息，所以此处需要通过编码方式加载，这可以通过 Configuration 对象的 addClass()方法实现，具体代码如下。

```
config.addClass(BranchForm.class);
```

addClass()方法用于加载实体类。

(3) 完成配置文件和映射文件的加载后，将得到一个包括所有 Hibernate 运行期参数的 Configuration 实例，通过 Configuration 实例的 buildSessionFactory()方法可以构建一个惟一的 SessionFactory，代码如下。

```
SessionFactory sessionFactory = config.buildSessionFactory();
```

Session: 一般的持久化方法（CRUD）都是通过 Session 来调用的，Session 是非线程安全的。

Session 的创建与关闭：Session 是一个轻量级对象，通常将每个 Session 实例和一个数据库事务绑定，也就是每执行一个数据库事务，都应该先创建一个新的 Session 实例，在使用 Session 后，还需要关闭 Session。

Session 的创建

创建 SessionFactory 后，就可以通过 SessionFactory 创建 Session 实例，通过 SessionFactory 创建 Session 实例的代码如下。

```
Session session=sessionFactory.openSession();
```

创建 Session 后，就可以通过创建的 Session 进行持久化操作了。

Session 的关闭

在创建 Session 实例后，不论是否执行事务，最后都需要关闭 Session 实例，释放 Session 实例占用的资源。

关闭 Session 实例的代码如下：

```
session.close();
```

下面来看一下：**getCurrentSession** 与 **openSession()** 的区别

1. **getCurrentSession** 创建的 session 会和绑定到当前线程,而 **openSession** 不会。

2 `getCurrentSession` 创建的线程会在事务回滚或事物提交后自动关闭,而
`openSession` 必须手动关闭

3. `getCurrentSession ()` 使用当前的 `session`, `openSession()` 重新建立一个
新的 `session`

这里 `getCurrentSession` 本地事务(本地事务:jdbc)时 要在配置文件里进行
如下设置

* 如果使用的是本地事务 (jdbc 事务)

```
<property name="hibernate.current_session_context_class">thread</prop  
erty>
```

* 如果使用的是全局事务 (jta 事务)

```
<property name="hibernate.current_session_context_class">jta</property  
>
```

`openSession()` 与 `getCurrentSession()` 有何不同和关联呢?

在 `SessionFactory` 启动的时候, `Hibernate` 会根据配置创建相应
的 `CurrentSessionContext`, 在 `getCurrentSession()` 被调用的时候, 实际被执
行的方法是 `CurrentSessionContext.currentSession()`。在 `currentSession()` 执
行时, 如果当前 `Session` 为空, `currentSession` 会调
用 `SessionFactory` 的 `openSession`。所以 `getCurrentSession()` 对
于 `Java EE` 来说是更好的获取 `Session` 的方法。

事务 transaction: `Hibernate` 是对 `JDBC` 的轻量级对象封装, `Hibernate` 本身
是不具备 `Transaction` 处理功能的, `Hibernate` 的 `Transaction` 实际上是底层的
`JDBC Transaction` 的封装, 或者是 `JTA Transaction` 的封装, 下面我们详细的
分析:

Hibernate 可以配置为 JDBCTransaction 或者是 JTATransaction，这取决于你在 hibernate.properties 或者 hibernate.cfg.xml 中的配置，如果你什么都不配置，默认情况下使用 JDBCTransaction，如果你配置

为：hibernate.transaction.factory_class =net.sf.hibernate.transaction.JTATransactionFactory

将使用 JTATransaction 。

JDBCTransaction 究竟是什么东西呢？来看看源代码就清楚了：

Hibernate3.3.2 源代码中的类 org.hibernate.transaction.JDBCTransaction:

[java] [view plain](#)[copy](#)[print](#)?

```
1. public void begin() throws HibernateException {
2.
3.  ...
4.
5.  try {
6.
7.    toggleAutoCommit= jdbcContext.connection().getAutoCommit();
8.
9.    if ( log.isDebugEnabled() ) {
10.
11.      log.debug("currentautocommitstatus: " + toggleAutoCommit);
12.
13.    }
14.
15.    if (toggleAutoCommit) {
16.
17.      log.debug("disabling autocommit");
18.
19.      jdbcContext.connection().setAutoCommit(false);
20.
21.    }
```

```
22.  
23.} ... }
```

这是启动 Transaction 的方法，看到 `connection().setAutoCommit(false)` 了吗？是不是很熟悉？ 再来看

[java] [view plaincopyprint?](#)

```
1. public void commit() throws HibernateException {  
2.  
3. ...  
4.  
5. try {  
6.  
7. commitAndResetAutoCommit();  
8.  
9. log.debug("committed JDBC Connection");  
10.  
11.committed = true;  
12.  
13.if ( callback ) {  
14.  
15.jdbcContext.afterTransactionCompletion( true, this );  
16.  
17.}  
18.  
19.notifyLocalSynchsAfterTransactionCompletion( Status.STATUS_COMMITTED );  
20.  
21.}... ; }  
22.  
23.commitAndResetAutoCommit 方法源码：  
24.  
25.private void commitAndResetAutoCommit() throws SQLException {  
26.  
27.try {
```



```

28.
29.jdbcContext.connection().commit();
30.
31.}
32.
33.finally {
34.
35.toggleAutoCommit();
36.
37.}}

```

这是提交方法，看到 `connection().commit()` 了吗？下面就不用我多说了，这个类代码非常简单易懂，通过阅读使我们明白 Hibernate 的 Transaction 都在干了些什么？我现在把用 Hibernate 写的例子翻译成 JDBC，大家就一目了然了：

```

Connection conn = ...; <--- session = sf.openSession();
conn.setAutoCommit(false); <--- tx = session.beginTransaction(); ... <--- ...
conn.commit(); <--- tx.commit(); (对应左边的两句)
conn.setAutoCommit(true); conn.close(); <--- session.close();

```

看明白了吧，Hibernate 的 JDBCTransaction 根本就是 `conn.commit` 而已，根本毫无神秘可言，只不过在 Hibernate 中，Session 打开的时候，就会自动 `conn.setAutoCommit(false)`，不像一般的 JDBC，默认都是 `true`，所以你最后不写 `commit` 也没有关系，由于 Hibernate 已经把 `AutoCommit` 给关掉了，所以用 Hibernate 的时候，你在程序中不写 Transaction 的话，数据库根本就没有反应。

（五十三）细谈 Hibernate （四）Hibernate 常用配置文件详解

初学 hibernate 的童鞋，刚开应该都有这种感觉，hibernate 的配置文件好麻烦，还不如 jdbc 访问数据库呢，直接写代码，多方便，用 hibernate 还要写代码，还要写配置，太麻烦了。至少我刚开始学习的时候就是这么想的。配置文件确实有他枯燥的一面，但等你真正深入学习的时候，你就可以发现他枯燥的背后却藏着很多强大的功能，呵呵，让我说的这么玄乎，那就让我们一起来看看吧，让我们一起来见证一下这些配置文件的强大。

Hibernate 中配置主要分为两种：一种包含了 Hibernate 与数据库的基本连接信息，在 Hibernate 工作的初始阶段，这些信息被先后加载到 Configuration 和 SessionFactory 实例；另一种包含了 Hibernate 的基本映射信息，即系统中每一个类与其对应的数据库表之间的关联信息，在 Hibernate 工作的初始阶段，这些信息通过 hibernate.cfg.xml 的 mapping 节点被加载到 Configuration 和 SessionFactory 实例。这两种文件信息包含了 Hibernate 的所有运行期参数。下面我们用详细的例子来说明这两种文件的基本结构和内容。

实现包含了 Hibernate 与数据库的基本连接信息的配置方式有两种方式：

第一种是使用 hibernate.properties 文件作为配置文件。

第二种是使用 hibernate.cfg.xml 文件作为配置文件。

1. 使用 hibernateproperties 作为配置文件

对于 hibernate.properties 作为配置文件的方式，比较适合于初学者。因为初学者往往很难记住 xml 配置文件的格式，以及需要配置哪些属性。在

Hibernate 发布包的 etc 路径下，提供了一个 hibernate.properties 文件，该文件列出了 Hibernate 的所有属性。每个配置段都给出了大致的注释，用户只要取消所需配置段的注释，就可以快速配置 Hibernate 和数据库的链接此处给出使用 hibernate.properties 文件创建 Configuration 对象的方法。

//实例化 configuration 对象

```
Configuration cfg = new Configuration()
```

//多次调用 addResource()方法，添加映射文件

```
cfg.addResource("Item.hbm.xml")
```

```
cfg.addResource("Bid.hbm.xml");
```

查看 hibernate.properties 文件发现，该文件没有提供 Hibernate 映射文件的方式。因此使用 hibernate.properties 文件来作为配置文件时，必须使用 Configuration 的.addResource()方法，使用该方法来添加映射文件。

注意：正如上面的代码所示，使用 hibernate.properties 文件配置 Hibernate 的属性固然简单，但是因为要手动添加映射文件，当映射文件极其多时，这是一件非常催人泪下的事情。这也就是在实际开发中，不常使用 hibernate.properties 文件作为配置文件的原因。

当然还有另一种添加配置文件的策略，因为映射文件和持久化类是一一对应的，可以通过 Configuration 对象来添加持久化类，让 Hibernate 自己来搜索映射文件。

//实例化 configuration 对象

```
Configuration cfg = new Configuration()
```

//多次调用 addClass()方法，直接添加持久化类

```
cfg.addClass(ppp.Item.class)
cfg.addClass(ppp.Bld.class);
```

2. 使用 **hibernate.cfg.xml** 作为配置文件

关于 xml 配置形式，我感觉也没必要多说什么。下面的一个例子足以把这种配置说明清楚，下面我们一起来看一个带有详细注释的

hibernate.cfg.xml 文件：

[html] [view plaincopyprint?](#)

1. <!--标准的 XML 文件的起始行，**version=** 表明 XML 的版本，**encoding=**
表
- 2.
3. 明 XML 文件的编码方式-->
- 4.
5. **<?xml version='1.0' encoding='gb2312' ?>**
- 6.
7. <!--表明解析本 XML 文件的 DTD 文档位置，DTD 是 DocumentType Definition 的缩写，
- 8.
9. 即文档类型的定义，XML 解析器使用 DTD 文档来检查 XML 文件的合法性。
- 10.
- 11.hibernate.sourceforge.net/hibernate-configuration-3.0dtd 可以在
- 12.
- 13.Hibernate3.1.3 软件包中的 src\org\hibernate 目录中找到此文件-->
- 14.
- 15.<!DOCTYPE hibernate-configuration PUBLIC
- 16.
17. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
- 18.
19. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
- 20.

21. <!--声明 Hibernate 配置文件的开始-->

22.

23. **<hibernate-configuration>**

24.

25. <!--表明以下的配置是针对 session-factory 配置的，SessionFactory 是

26.

27. Hibernate 中的一个类，这个类主要负责保存 Hibernate 的配置信息，以及对 Session

的

28.

29. 操作-->

30.

31. **<session-factory>**

32.

33. <!--配置数据库的驱动程序，Hibernate 在连接数据库时，需要用到数据库的驱

34.

35. 动程序-->

36.

37. **<property name=** **>**com.mysql.jdbc.Driv

er**</property>**

38.

39. <!--设置数据库的连接 url:jdbc:mysql://localhost/hibernate,其中 localhost 表示

mysql 服务器名称，此处为本机，hibernate 是数据库名-->

40.

41. **<property name=** **>** jdbc:mysql://localhost/hiberna

te**</property>**

42.

43.

44.

45. <!--连接数据库是用户名-->

46.

47. **<property name=** **>**root**</property>**

48.

49. <!--连接数据库是密码-->

50.

[illegible]

79. <!--jdbc.use_scrollable_resultset 是否允许 Hibernate 用 JDBC 的可滚动的结果集。
对分页的结果集。对分页时的设置非常有帮助-->

80.

81. <propertynamepropertyname=
 >false</prope
 rty>

82.

83.

84.

85. <!--connection.useUnicode 连接数据库时是否使用 Unicode 编码-->

86.

87. <propertynamepropertyname=
 >true</property>

88.

89.

90.

91. <!--connection.characterEncoding 连接数据库时数据的传输字符集编码方式，最
好设置为 gbk，用 gb2312 有的字符不全-->

92.

93. <propertynamepropertyname=
 >gbk</prop
 erty>

94.

95.

96.

97. <!--hibernate.dialect 只是 Hibernate 使用的数据库方言,就是要用 Hibernate 连
接那种类型的数据库服务器。-->

98.

99. <property name=
 >

100.

101. org.hibernate.dialect.MySQLDialect</property>

102.

103. <!--是否自动创建数据库表 他主要有一下几个值:

104.

105. validate:当 sessionFactory 创建时，自动验证或者 schema 定义导入数据库。

106.

107. create:每次启动都 drop 掉原来的 schema，创建新的。

108.

109. create-drop:当 sessionFactory 明确关闭时，drop 掉 schema。

110.

111. update(常用):如果没有 schema 就创建，有就更新。

112.

113. -->

114.

115. `<propertynamepropertyname=>create</property>`

116.

117.

118.

119. <!--配置此处 sessionFactory.getCurrentSession()可以完成一系列的工作，当调用时，

120. hibernate 将 session 绑定到当前线程，事务结束后，hibernate

121. 将 session 从当前线程中释放，并且关闭 session。当再次调用

getCurrentSession

122. ()时，将得到一个新的 session，并重新开始这一系列工作。-->

123.

124. `<propertynamepropertyname=>thread</pro
perty>`

125.

126.

127.

128. <!--指定映射文件为“hibernate/ch1/UserInfo.hbm.xml”-->

129.

130. `<mappingresourcemappingresource=>`

131.

132. `</session-factory>`

133.

134. `</hibernate-configuration>`

以上应该是大部分常用的配置文件属性，当然里面的很多部分都是可以在配置 **hibernate** 开发环境时自动生成的，刚开始的时候还是建议大家手动的去配置一下，可以达到熟悉的的时候在用自动生成

下面我们继续看一下包含了 **Hibernate** 的基本映射信息的配置文件，也就是系统中每一个类与其对应的数据库表之间的关联信息，这种配置文件一般命名为：类名.hbm.xml，下面我们通过一个具体的代码示例来看一下类名.hbm.xml 的结构：

[html] [view plaincopyprint?](#)

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6.
7.
8.
9. <hibernate-mapping>
10.
11. <class name="org.hibernate.test.nodatabase.SimpleEntity" table="SimpleEntity">
12.
13. <id name="id" column="id" type="int">
14.
15. <generator class="org.hibernate.id.SimpleGenerator"></generator>
16.
17. </id>
18.
```

```

19. <property name=      column=      type=      ></property>
20.
21. <property name=      column="cardId"      ="string"></property>
22.
23. <property name=      column=      type=      ></property>
24.
25. <set name=      table=      cascade=      >
26.
27. <key column=      ></key>
28.
29. <many-to-many class=      column=      >
30.
31. </many-to-many>
32.
33. </set>
34.
35. </class>
36.
37. </hibernate-mapping></SPAN>

```

下面我们一个个标签进行讲解一下：

1.hibernate-mapping

这个元素包括一些可选的属性。`schema` 和 `catalog` 属性，指明了这个映射所连接（refer）的表所在的 `schema` 和/或 `catalog` 名称。假若指定了这个属性，表名会加上所指定的 `schema` 和 `catalog` 的名字扩展为全限定名。假若没有指定，表名就不会使用全限定名。`default-cascade` 指定了未明确注明 `cascade` 属性的 Java 属性和集合类 Hibernate 会采取什么样的默认级联风格。`auto-import` 属性默认让我们在查询语言中可以使用非全限定名的类名。

[html] [view plaincopyprint?](#)

```

1. <SPAN xmlns=                                ><hibernate-mapping
2. schema=                                     (1)
3. catalog=                                   (2)
4. default-cascade=                           (3)
5. default-access=                             (4)
6. default-lazy=                               (5)
7. auto-import=                               (6)
8. package=                                   (7)
9. /></SPAN>

```

(1) schema (可选): 数据库 schema 的名称。

(2) catalog (可选): 数据库 catalog 的名称。

(3) default-cascade (可选 - 默认为 none): 默认的级联风格。

(4) default-access (可选 - 默认为 property): Hibernate 用来访问所有属性的策略。可以通过实现 PropertyAccessor 接口自定义。

(5) default-lazy (可选 - 默认为 true): 指定了未明确注明 lazy 属性的 Java 属性和集合类， Hibernate 会采取什么样的默认加载风格

(6) auto-import (可选 - 默认为 true): 指定我们是否可以在查询语言中使用非全限定的类名（仅限于本映射文件中的类）。

(7) package (可选): 指定一个包前缀，如果在映射文档中没有指定全限定的类名，就使用这个作为包名。

2.class

使用 class 元素来定义一个持久化类:

[html] [view plain](#) [copy](#) [print?](#)

```

1. <SPAN xmlns=                                ><class

```

```

2. name= (1)
3. table= (2)
4. discriminator-value= (3)
5. mutable= (4)
6. schema= (5)
7. catalog= (6)
8. proxy= (7)
9. dynamic-update= (8)
10. dynamic-insert= (9)
11. select-before-update= (10)
12. polymorphism= (11)
13. where= (12)
14. persister= (13)
15. batch-size= (14)
16. optimistic-lock= (15)
17. lazy= (16)
18. entity-name= (17)
19. check= (18)
20. rowid= (19)
21. subselect= (20)
22. abstract= (21)
23. node=
24. /></SPAN>

```

(1) **name** (可选): 持久化类（或者接口）的 **Java** 全限定名。如果这个属性不存在，**Hibernate** 将假定这是一个非 **POJO** 的实体映射。

(2) **table** (可选 - 默认是类的非全限定名): 对应的数据库表名。

(3) **discriminator-value** (可选 - 默认和类名一样): 一个用于区分不同的子类的值，在多态行为时使用。它可以接受的值包括 **null** 和 **not null**。

(4) **mutable** (可选，默认值为 **true**): 表明该类的实例是可变的或者不可变的。

(5) **schema** (可选): 覆盖在根<hibernate-mapping>元素中指定的 **schema** 名字。

(6) **catalog** (可选): 覆盖在根<hibernate-mapping>元素中指定的 **catalog** 名字。

(7) **proxy** (可选): 指定一个接口，在延迟装载时作为代理使用。你可以在这里使用该类自己的名字。

(8) **dynamic-update** (可选, 默认为 **false**): 指定用于 **UPDATE** 的 **SQL** 将会在运行时动态生成，并且只更新那些改变过的字段。

(9) **dynamic-insert** (可选, 默认为 **false**): 指定用于 **INSERT** 的 **SQL** 将会在运行时动态生成，并且只包含那些非空值字段。

(10) **select-before-update** (可选, 默认为 **false**): 指定 **Hibernate** 除非确定对象真正被修改了（如果该值为 **true**—译注），否则不会执行 **SQL UPDATE** 操作。在特定场合（实际上，它只在一个瞬时对象（**transient object**）关联到一个新的 **session** 中时执行的 **update()**中生效），这说明 **Hibernate** 会在 **UPDATE** 之前执行一次额外的 **SQL SELECT** 操作，来决定是否应该执行 **UPDATE**。

(11) **polymorphism**（多态）（可选, 默认值为 **implicit** (隐式)): 界定是隐式还是显式的使用多态查询（这只在 **Hibernate** 的具体表继承策略中用到—译注）。

(12) **where** (可选) 指定一个附加的 **SQL WHERE** 条件，在抓取这个类的对象时会一直增加这个条件。

(13) **persister** (可选): 指定一个定制的 **ClassPersister**。

(14) **batch-size** (可选,默认是 1) 指定一个用于根据标识符 (**identifier**) 抓取实例时使用的"**batch size**" (批次抓取数量)。

(15) **optimistic-lock** (乐观锁定) (可选, 默认是 **version**): 决定乐观锁定的策略。

(16) **lazy** (可选): 通过设置 **lazy="false"**, 所有的延迟加载 (**Lazyfetching**) 功能将被全部禁用 (**disabled**)。

(17) **entity-name** (可选, 默认为类名): **Hibernate3** 允许一个类进行多次映射 (前提是映射到不同的表), 并且允许使用 **Maps** 或 **XML** 代替 **Java** 层次的实体映射 (也就是实现动态领域模型, 不用写持久化类一译注)。

(18) **check** (可选): 这是一个 **SQL** 表达式, 用于为自动生成的 **schema** 添加多行 (**multi-row**) 约束检查。

(19) **rowid** (可选):**Hibernate** 可以使用数据库支持的所谓的 **ROWIDs**, 例如: **Oracle** 数据库, 如果你设置这个可选的 **rowid**, **Hibernate** 可以使用额外的字段 **rowid** 实现快速更新。**ROWID** 是这个功能实现的重点, 它代表了一个存储元组 (**tuple**) 的物理位置。

(20) **subselect** (可选): 它将一个不可变 (**immutable**) 并且只读的实体映射到一个数据库的子查询中。当你想用视图代替一张基本表的时候, 这是有用的, 但最好不要这样做。更多的介绍请看下面内容。

(21) **abstract** (可选): 用于在<**union-subclass**>的继承结构 (**hierarchies**) 中标识抽象超类。

3.id

被映射的类必须定义对应数据库表主键字段。大多数类有一个 **JavaBeans** 风格的属性，为每一个实例包含唯一的标识。`<id>` 元素定义了该属性到数据库表主键字段的映射。

[\[html\] view plaincopyprint?](#)

```
1. <id
2.   name=           (1)
3.   type=           (2)
4.   column=         (3)
5.   unsaved-value=   (4)
6.   access=         (5)
7.
8.   length=         (6)
9.
10. <generatorclassgeneratorclass=
11. </id>
```

(1) **name** (可选): 标识属性的名字。

(2) **type** (可选): 标识 **Hibernate** 类型的名字。(如果没配置,**hibernate** 将会自动转化成相应的数据库类型)

(3) **column** (可选 - 默认为属性名): 主键字段的名称。

(4) **unsaved-value** (可选 - 默认为一个切合实际 (**sensible**) 的值): 一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。这可以把这种实例和从以前的 **session** 中装载过 (可能又做过修改--译者注) 但未再次持久化的实例区分开来。

(5) **access** (可选 - 默认为 **property**): **Hibernate** 用来访问属性值的策略。

(6)length="L"指定长度

4. Generator

可选的<generator>子元素是一个 Java 类的名字，用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数，用<param>元素来传递。

[html] view plaincopyprint?

```
1. <id name="id"      ="long" column=      >
2.   <generatorclassgeneratorclass=      >
3.     <paramnameparamname=      >uid_table</param>
4.     <paramnameparamname=      >next_hi_value_column</param>
5.   </generator>
6. </id>
```

所有的生成器都实现 org.hibernate.id.IdentifierGenerator 接口。这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然，Hibernate 提供了很多内置的实现。

下面是一些内置生成器的快捷名字：

increment

用于为 long, short 或者 int 类型生成唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。在集群下不要使用。

identity

对 DB2,MySQL, MS SQL Server, Sybase 和 HypersonicSQL 的内置标识字段提供支持。返回的标识符是 long, short 或者 int 类型的。 sequence

在 DB2, PostgreSQL, Oracle, SAP DB, McKoi 中使用序列 (**sequence**), 而在 Interbase 中使用生成器(**generator**)。返回的标识符是 **long**, **short** 或者 **int** 类型的。

hilo

使用一个高/低位算法高效的生成 **long**, **short** 或者 **int** 类型的标识符。给定一个表和字段 (默认分别是 **hibernate_unique_key** 和 **next_hi**) 作为高位值的来源。高/低位算法生成的标识符只在一个特定的数据库中是唯一的。

seqhilo

使用一个高/低位算法来高效的生成 **long**, **short** 或者 **int** 类型的标识符, 给定一个数据库序列 (**sequence**) 的名字。

uuid

用一个 **128-bit** 的 **UUID** 算法生成字符串类型的标识符, 这在一个网络中是唯一的 (使用了 **IP** 地址)。UUID 被编码为一个 **32** 位 **16** 进制数字的字符串。

guid

在 **MS SQL Server** 和 **MySQL** 中使用数据库生成的 **GUID** 字符串

native

根据底层数据库的能力选择 **identity**, **sequence** 或者 **hilo** 中的一个

assigned

让应用程序在 **save()** 之前为对象分配一个标示符。这是 **<generator>** 元素没有指定时的默认生成策略。手动分配主键的时候要设置成它

select

通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。

foreign

使用另外一个相关联的对象的标识符。通常和 **<one-to-one>** 联合起来使用。

5. property

<property>元素为类定义了一个持久化的,JavaBean 风格的属性。

[html] view plaincopyprint?

```
1. <SPAN xmlns=><property
2.   name=                (1)
3.   column=              (2)
4.   type=                (3)
5.   update=              (4)
6.   insert=              (4)
7.   formula=             (5)
8.   access=              (6)
9.   lazy=                (7)
10.  unique=              (8)
11.  not-null=            (9)
12.  optimistic-lock=     (10)
13.  generated=           (11)
14.  node=
15.
16.  index=
17.  unique_key=
18.  length=
19.  precision=
20.  scale=
21. />
22. </SPAN>
```

(1) name: 属性的名字,以小写字母开头。

(2) column (可选 - 默认为属性名字): 对应的数据库字段名。也可以通过嵌套的<column>元素指定。

(3) **type** (可选): 一个 **Hibernate** 类型的名字。

(4) **update, insert** (可选 - 默认为 **true**): 表明用于 **UPDATE** 和/或 **INSERT** 的 **SQL** 语句中是否包含这个被映射了的字段。这二者如果都设置为 **false** 则表明这是一个“外源性 (**derived**)”的属性，它的值来源于映射到同一个 (或多个) 字段的某些其他属性，或者通过一个 **trigger**(触发器) 或其他程序生成。

(5) **formula** (可选): 一个 **SQL** 表达式，定义了这个计算 (**computed**) 属性的值。计算属性没有和它对应的数据库字段。

(6) **access** (可选 - 默认值为 **property**): **Hibernate** 用来访问属性值的策略。

(7) **lazy** (可选 - 默认为 **false**): 指定指定实例变量第一次被访问时，这个属性是否延迟抓取 (**fetched lazily**) (需要运行时字节码增强)。

(8) **unique** (可选): 使用 **DDL** 为该字段添加唯一的约束。同样，允许它作为 **property-ref** 引用的目标。

(9) **not-null** (可选): 使用 **DDL** 为该字段添加可否为空 (**nullability**) 的约束。

(10) **optimistic-lock** (可选 - 默认为 **true**): 指定这个属性在做更新时是否需要获得乐观锁定 (**optimistic lock**)。换句话说，它决定这个属性发生脏数据时版本 (**version**) 的值是否增长。

(11) **generated** (可选 - 默认为 **never**): 表明此属性值是否实际上是由数据库生成的。

typename 可以是如下几种:

Hibernate 基本类型名 比如: **integer, string, character, date, timestamp, float,**

binary, serializable, object, blob)。

一个 Java 类的名字，这个类属于一种默认基础类型（比如： int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob）。

一个可以序列化的 Java 类的名字。

一个自定义类型的类的名字。（比如： com.illflow.type.MyCustomType）。

基本值类型(Hibernate 内建立自己的类型,从 java 转化成数据库类型) string:
从 java.lang.String 到 VARCHAR (或者 Oracle 的 VARCHAR2)的映射。

date, time, timestamp: 从 java.util.Date 和其子类到 SQL 类型 DATE, TIME
和 TIMESTAMP (或等价类型)的映射。

calendar, calendar_date: 从 java.util.Calendar 到 SQL 类型 TIMESTAMP
和 DATE(或等价类型)的映射。

big_decimal, big_integer: 从 java.math.BigDecimal 和 java.math.BigInteger
到 NUMERIC (或者 Oracle 的 NUMBER 类型)的映射。

locale, timezone, currency : 从 java.util.Locale, java.util.TimeZone 和
java.util.Currency 到 VARCHAR (或者 Oracle 的 VARCHAR2 类型)的映射。
Locale 和 Currency 的实例被映射为它们的 ISO 代码。TimeZone 的实例被
映射为它的 ID。

Class: 从 java.lang.Class 到 VARCHAR (或者 Oracle 的 VARCHAR2 类
型)的映射。Class 被映射为它的全限定名。

Binary: 把字节数组(byte arrays)映射为对应的 SQL 二进制类型。

Text: 把长 Java 字符串映射为 SQL 的 CLOB 或者 TEXT 类型。

Serializable: 把可序列化的 Java 类型映射到对应的 SQL 二进制类型。你也可以为一个并非默认为基本类型的可序列化 Java 类或者接口指定 Hibernate 类型 `serializable`。

clob, blob: JDBC 类 `java.sql.Clob` 和 `java.sql.Blob` 的映射。某些程序可能不适合使用这个类型，因为 `blob` 和 `clob` 对象可能在一个事务之外是无法重用的。(而且，驱动程序对这种类型的支持充满着补丁和前后矛盾。)

（五十四）细谈 Hibernate（五）Hibernate 一对多关系映射

前几篇系列博客：

[细谈 Hibernate（一）hibernate 基本概念和体系结构](#)

[细谈 Hibernate（二）开发第一个 hibernate 基本详解](#)

[细谈 Hibernate（三）Hibernate 常用 API 详解及源码分析](#)

[细谈 Hibernate（四）Hibernate 常用配置文件详解](#)

在前几篇博客，我们初步对 Hibernate 有了一定的基础性的认知了，也能够简单的用 hibernate 进行增删改查，但 hibernate 真正的难度和精髓我们都还没接触到，其中最主要的关联映射就是其中一个，这篇博客，我们就一起来看一下这个 hibernate 关联映射。我们大家都知道，在域模型(实体域)中，关联关系是类与类之间最普遍的关系，他是指通过一个对象持有另一个对象的实例根据 UML 语言，关系是有方向的。实质上**关联映射的本质：将关联关系映射到数据库，所谓的关联关系是对象模型在内存中的一个或多个引用**。搞清关联映射的关键就在于搞清实体之间的关系。下面我们首先来看一下具体什么事关联关系：

一：关联关系

1.关联关系的方向可分为单向关联和双向关联。

单向关联：假设存在两张表 person 表和 address 表，如果在应用的业务逻辑中，仅需要每个 person 实例能够查询得到其对应的 Address 实例，而 Address 实例并不需要查询得到其对应的 person 实例；或者反之。

双向关联：既需要每个 person 实例能够查询得到其对应的 Address 实例，Address 实例也需要查询得到其对应的 person 实例。

2.关联的数量，根据拥有被关联对象的个数确定

多对一（many to one）：如用户和组 学生和班级

一对多（one to many）：如用户和电子邮件

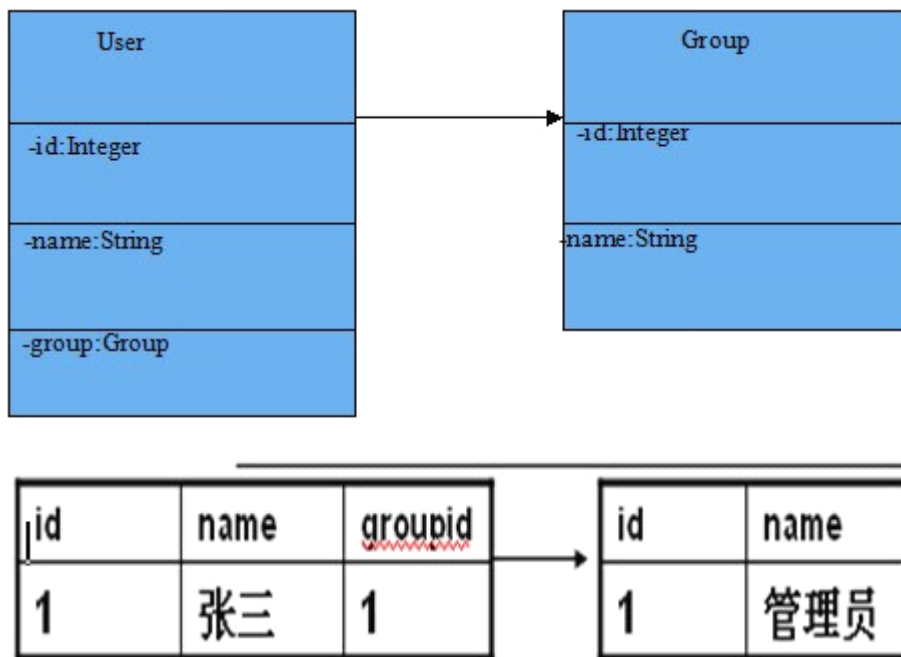
多对多（many to many）：如学生选课

一对一（one to one）：如用户和身份证

下面我们就开始讲第一种关联关系：一对多，一对多总共分为：单向一对多，单向多对一，双向一对多。这主要是站在关系双方各自角度来定义的。下面我们就来一一来看一下

二．单向多对一

单向多对一关联是最常见的单向关联关系,如：多个用户属于同一个组，多个学生处于同一个班级。之所以叫他多对一而不是一对多，是因为他们之间的关系是多的一方来维护的，下面我们就以多个用户属于同一个组来详细说明一下单向多对一。首先看一下他们的关系示例：



从上边的图示中可以看出，多个用户属于一个组，我们用多的一方来维护，所以我们可以根据用户可以知道他在哪个组，而不需要知道一个组里有哪些学生，这就是所谓的单向的。多对一映射原理：在多的一端加入一个外键指向一的一端，它维护的关系多指向一，一对多映射原理，在多的一端加入一个外键指向一的一端，她维护的关系是一指向多，也就是说一对多与多对一的映射原理是一样的，只是站的角度不一样。下面来看一下单向多对一关系配置文件：

Group: 一的一方，不需要维护关系，所以和普通配置一样

[html] view plaincopyprint?

1. `<hibernate-mapping>`
- 2.
3. `<class name=` table="group">
- 4.
5. `<id name=` >
- 6.

```

7.     <column name=      />
8.
9.     <generator class=      />
10.
11. </id>
12.
13. <property name=      type=      >
14.
15.     <column name=      length=      not-null=      />
16.
17. </property>
18.
19. </class>
20.
21. </hibernate-mapping>

```

User: 多的一方，需要维护双方关系，内有一的一方引用：

[html] [view plaincopyprint?](#)

```

1. <hibernate-mapping>
2.
3. <class name=      table="user">
4.
5.     <id name=      >
6.
7.         <column name=      />
8.
9.         <generator class=      />
10.
11.     </id>
12.
13.     <many-to-one name="group" column="group_id" />
14.
15. </many-to-one>

```

```

16.
17.     <property name=         type=         >
18.
19.     <column name=         length=         not-null=         />
20.
21. </property>
22.
23. </class>
24.
25.</hibernate-mapping>

```

注：1.many-to-one 元素的常用属性：

属性	含义和作用	必须	默认值
name	映射类属性的名称	Y	
class	关联类的完全限定名	N	
column	关联的字段	N	
not-null	设置关联的字段的值是否可以为空	N	false
lazy	指定关联对象是否使用延迟加载以及延迟加载的策略	N	proxy
fetch	设置抓取数据的策略	N	select

具体的抓取数据策略会再以后详细讲解

2.重要属性 — cascade（级联）

级联的意思是指定两个对象之间的操作联动关系，对一个对象执行了操作之后，对其指定的级联对象也需要执行相同的操作

总共可以取值为：all、none、save-update、delete

all-代表在所有的情况下都执行级联操作

none-在所有情况下都不执行级联操作

save-update-在保存和更新的时候执行级联操作

delete-在删除的时候执行级联操作

三. 单向一对多

所谓单向一对多，就是实体之间的关系由“一”的一端加载“多”的一端，关系由“一”的一端来维护，在 **JavaBean** 中是在“一”的一端中持有“多”的一端的集合，**Hibernate** 把这种关系反映到数据库的策略是在“多”的一端的表上加一个外键指向“一”的一端的表的主键。比如 **Class**（班级）和 **Student**（学生）之间是一对多的关系。一个班级里面有很多的学生，站在班级的角度上来看就是一个班级对应多个学生。我们来看一下具体的关系图：



从图上我们可以看出，在一的一端含有一个多的引用的集合，我们可以根据班级找到它有哪些学生，而不能根据学生找到他对应的班级。在一的一端维护的缺点：

- * 如果将 **student** 表里的 **classesid** 设为非空，则无法保存；
- * 因为不是在 **student** 端维护数据，所以 **student** 端不知道学生是哪个班的
- * 需要发出多余的 **update** 语句来更新关系；

下面我们就具体来看一下一对多中维护关系的“一”中是怎样来配置的：

Class 映射文件

[html] [view plaincopyprint?](#)

```
1. <hibernate-mapping>
2.
3.   <class name=                table="tb_class">
4.
5.     <id name=      >
6.
7.       <generator class="native"/>
8.
9.     </id>
10.
11.    <property name=      />
12.
13.    <set name=      >
14.
15.      <key column=      ></key>
16.
17.      <one-to-many class="cn.edu.bzu.hibernate.Student" />
18.
19.    </set>
20.
21.  </class>
22.
23. </hibernate-mapping>
```

注意：

1.<set>元素的 inverse 属性：在映射一对多的双向关联时，应该在“one”方把 inverse 属性设为 true，这样可提高应用性能。

inverse：控制反转，为 true 表示反转，由它方负责；反之，不反转，自己负

责；如果不设，one 和 many 两方都要负责控制，因此，会引发重复的 sql 语句以及重复添加数据，

2.级联删除(从数据库删除相关表记录)

当删除 **Customer** 对象时，及联删除 **Order** 对象.只需将 **cascad** 属性设为 **delete** 即可.

注：删除后的对象，依然存在于内存中，只不过由持久化态变为临时态.

3.父子关系(逻辑删除，只是解除了关联关系)

自动删除不再和 **Customer** 对象关联的 **Order** 对象.只需将 **cascade** 属性设为 **delete-orphan**.

注：当关联双方都存在父子关系，就可以把父方的 **cascade** 属性设为 **delete-orphan**，所谓父子关系，是由父方来控制子方的生命周期.

4.<set name="students" >

<key column="classid" ></key>

<one-to-many class= "cn.edu.bzu.hibernate.Student" />

</set>

Name 为--持久化对象的集合的属性名称

<key column="classid" ></key>外键的名称

<one-to-many class= "cn.edu.bzu.hibernate.Student" />持久化类

四：双向一对多关联

所谓双向一对多关联，同时配置单向一对多和单向多对一就成了双向一对多关联，上面两种都是单向的，但是在实际开发过程中，很多时候都是需要双向关联的，它在解决单向一对多维护关系的过程中存在的缺陷起了一定的修补作用。在插入学生的时候，如果班级不能为空，则学生是插入不了的。

还有如果插入成功，在开始解决班级字段是空的，在事务提交阶段，班级需要更新每一个学生的班级 ID，这样会产生大量的 Update 语句。影响效率。

所以一对多关系大多使用双向一对多映射。具体配置文件：

多的一方：Student 映射文件

[\[html\] view plaincopyprint?](#)

```
1. <hibernate-mapping>
2.
3.   <class name=           >
4.
5.     <id name=   >
6.
7.       <generator class="native"/>
8.
9.     </id>
10.
11.    <property name=       />
12.
13.    <many-to-one name="classes" column="classid"/>
14.
15. </class>
16.
17. </hibernate-mapping>
```

一的一方：Class 映射文件

[\[html\] view plaincopyprint?](#)

```
1. <hibernate-mapping>
2.
3.   <class name=           table="tb_class">
4.
5.     <id name=   >
6.
```

```

7.      <generator class="native"/>
8.
9.      </id>
10.
11.     <property name=          />
12.
13.     <set name=          >
14.
15.     <key column=          ></key>
16.
17.     <one-to-many class="cn.edu.bzu.hibernate.Student" />
18.
19.     </set>
20.
21. </class>
22.
23. </hibernate-mapping>

```

注：

1.一对多双向关联映：

* 在一的一端的集合上使用<key>,在对方表中加入一个外键指向一的一端；

* 在多一端采用<many-to-one>

2.key 标签指定的外键字段必须和<many-to-one>指定的外键字段一致，否则引用字段错误；

3.如果在一的一端维护一对多关联关系，hibernate 会发出多余的 Update 语句，多以我们一般在多的一端维护关联关系

4.关于 inverse 属性；

inverse 主要用在一对多，多对多双向关联上，inverse 可以设置到<set>集

合上，

默认 **inverse** 为 **false**，所以我们可以从一的一端和多的一端来维护关联关系，如果 **inverse** 为 **true**，我们只能从多的一端来维护关联关系，注意：**inverse** 属性，只影响存储（使存储方向转变），即持久，

5.区分 **inverse** 和 **cascade**

Inverse: 负责控制关系，默认为 **false**，也就是关系的两端都能控制，但这样会造成一些问题，更新的时候会因为两端都控制关系，于是重复更新。一般来说有一端要设为 **true**。

Cascade: 负责控制关联对象的级联操作，包括更新、删除等，也就是说对一个对象进行更新、删除时，其它对象也受影响，比如我删除一个对象，那么跟它是多对一关系的对象也全部被删除。

举例说明区别:删除“一”那一端一个对象 **O** 的时候,如果“多”的那一端的 **Inverse** 设为 **true**,则把“多”的那一端所有与 **O** 相关联的对象外键清空;如果“多”的那一端的 **Cascade** 设为 **Delete**,则把“多”的那一端所有与 **O** 相关联的对象全部删除。

（五十五）细谈 Hibernate （六）Hibernate 继承关系映射

在面向对象的程序领域中，类与类之间是有继承关系的，例如 Java 世界中只需要 **extends** 关键字就可以确定这两个类的父子关系，但是在关系数据库的世界中，表与表之间没有任何关键字可以明确指明这两张表的父子关系，表与表是没有继承关系这样的说法的。为了将程序领域中的继承关系反映到数据中，Hibernate 为我们提供了 3 中方案：

第一种方案：一个子类对应一张表。

第二种方案：使用一张表表示所有继承体系下的类的属性的并集。

第三种方案：每个子类使用一张表只存储它特有的属性，然后与父类所对应的表以一对一主键关联的方式关联起来。

这三种方案用官方的语言来说就是：

TPS：每个子类一个表(table per subclass)。

TPH:每棵类继承树使用一个表(table per class hierarchy)

TPC:类表继承。每个具体类一个表(table per concrete class) (有一些限制)

现在我们就根据一个实例来看一下这三种方案的各自优缺点，一起来熟悉一下这三种方案。现在假设有 People、Student、Teacher 三个类，父类为 People，Student 与 Teacher 为 People 的父类，代码如下：

People 类：

[java] [view plaincopyprint?](#)

```

1. public class People
2.
3. {
4.
5.     /*父类所拥有的属性*/
6.
7.     private Stringid;
8.
9.     private Stringname;
10.
11.    private Stringsex;
12.
13.    private Stringage;
14.
15.    private Timestampbirthday;
16.
17.    /*get 和 set 方法*/
18.
19.}

```

Student 类:

[java] [view plaincopyprint?](#)

```

1. public class Student extends People
2.
3. {
4.
5.     /*学生独有的属性*/
6.
7.     private String cardId;//学号
8.
9.     public String getCardId()
10.
11.    { return cardId;}

```

```
12.  
13. public void setCardId(String cardId)  
14.  
15. {  
16.  
17.     this.cardId = cardId;  
18.  
19. }}
```

Teacher 类:

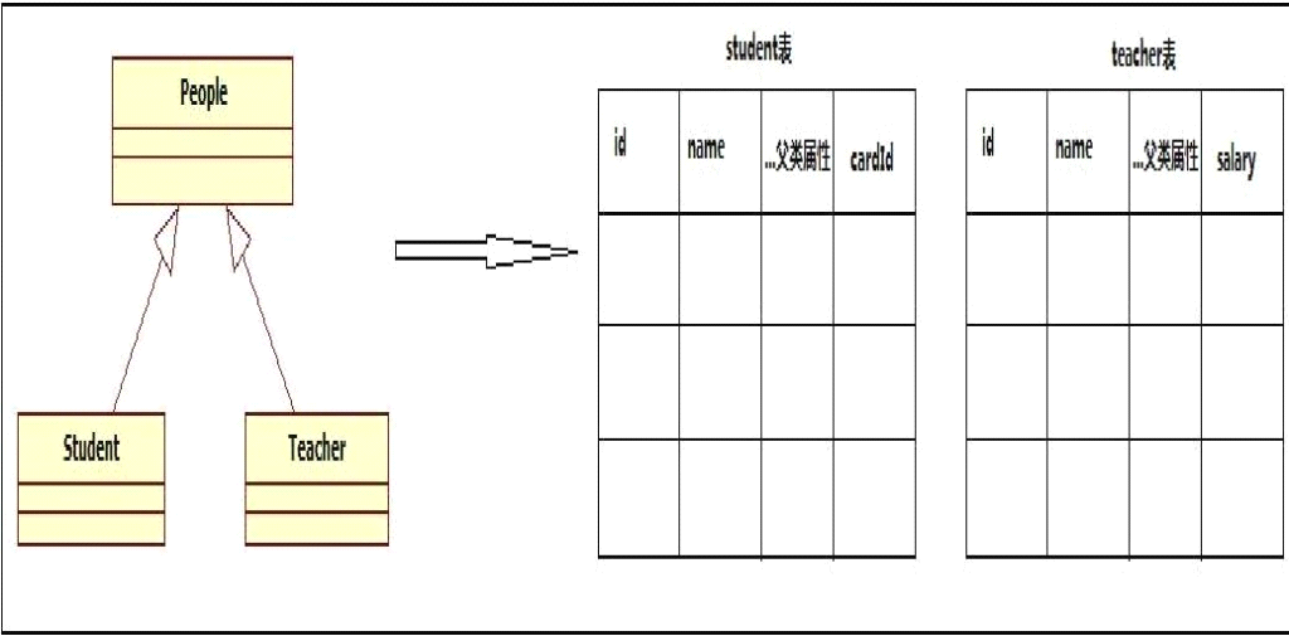
[java] [view plaincopyprint?](#)

```
1. public class Teacher extends People  
2.  
3. {  
4.  
5.     /*Teacher 所独有的属性*/  
6.  
7.     private int salary; //工资  
8.  
9.     public int getSalary()  
10.  
11. {  
12.  
13.     return salary;  
14.  
15. }  
16.  
17. public void setSalary(int salary)  
18.  
19. {  
20.  
21.     this.salary = salary;  
22.
```

- 23. }
- 24.
- 25.}

第一种方案：一个子类对应一张表 (TPS)

该方案是使继承体系中每一个子类都对应数据库中的一张表。示意图如下：



每一个子类对应的数据库表都包含了父类的信息，并且包含了自己独有的属性。每个子类对应一张表，而且这个表的信息是完备的，即包含了所有从父类继承下来的属性映射的字段。这种策略是使用<union-subclass>标签来定义子类的。

配置 People.hbm.xml 文件：

[html] [view plaincopyprint?](#)

- 1. `<?xml version="1.0" ="utf-8"?>`
- 2.

```

3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.
   0//EN""http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4.
5.
6.
7. <hibernate-mapping>
8.
9.   <class name="com.bzu.hibernate.pojos.People"           ="true">
10.
11.     <id name=      type=      >
12.
13.       <column name=      ></column>
14.
15.       <generator class=      ></generator>
16.
17.     </id>
18.
19.
20.
21.     <property name=      column="name"      ="string"></property>
22.
23.     <property name=      column="sex"      ="string"></property>
24.
25.     <property name=      column="age"      ="string"></property>
26.
27.     <property name=      column="birthday"      ="timestamp"></property>
28.
29.
30.
31.     <!--
32.
33.     <union-subclass name="com.bzu.hibernate.pojos.Student"      ="student">
34.
35.       <property name=      column="cardId"      ="string"></property>

```

```

36.
37.     </union-subclass>
38.
39.     <union-subclass name="com.bzu.hibernate.pojos.Teacher"         ="teacher">

40.
41.         <property name=             column="salary"         ="integer"></property>
42.
43.     </union-subclass>
44.
45. -->
46.
47. </class>
48.
49. <union-subclass name=
50.
51.     table="student"         ="com.bzu.hibernate.pojos.People">
52.
53.     <property name=             column="cardId"         ="string"></property>
54.
55. </union-subclass>
56.
57.
58.
59. <union-subclass name=
60.
61.     table="teacher"         ="com.bzu.hibernate.pojos.People">
62.
63.     <property name=             column="salary"         ="integer"></property>
64.
65. </union-subclass>
66.
67. </hibernate-mapping>

```

以上配置是一个子类一张表方案的配置，<union-subclass>标签是用于指示出该 hbm 文件所表示的类的子类，如 People 类有两个子类，就需要两个<union-subclass>标签以此类推。<union-subclass>标签的"name"属性用于指定子类的全限定名称，"table"属性用于指定该子类对应的表的名称，"extends"属性用于指定该子类的父类，注意该属性与<union-subclass>标签的位置有关，若<union-subclass>标签作为<class>标签的子标签，则"extends"属性可以不设置，否则需要明确设置"extends"属性。<class>标签中的"abstract"属性如果值为 true 则，不会生成表结构。如果值为 false 则会生成表结构，但是不会插入数据。

根据 People.hbm.xml 生成表结构，可以看到一个子类对应一张表：

[sql] [view plaincopyprint?](#)

```
1.      <SPAN xmlns="http://www.w3.org/1999/xhtml">drop table if
      exists student
2.
3.      drop table if exists teacher
4.
5.      create table student (
6.
7.          id varchar(255) not null,
8.
9.          name varchar(255),
10.
11.          sex varchar(255),
12.
13.          age varchar(255),
14.
15.          birthday datetime,
```



```

16.
17.             cardId varchar(255),
18.
19.             primary key (id)
20.
21.         )
22.
23.     create table teacher (
24.
25.         id varchar(255) not null,
26.
27.         name varchar(255),
28.
29.         sex varchar(255),
30.
31.         age varchar(255),
32.
33.         birthday datetime,
34.
35.         salary integer,
36.
37.         primary key (id)
38.
39.     )</SPAN>

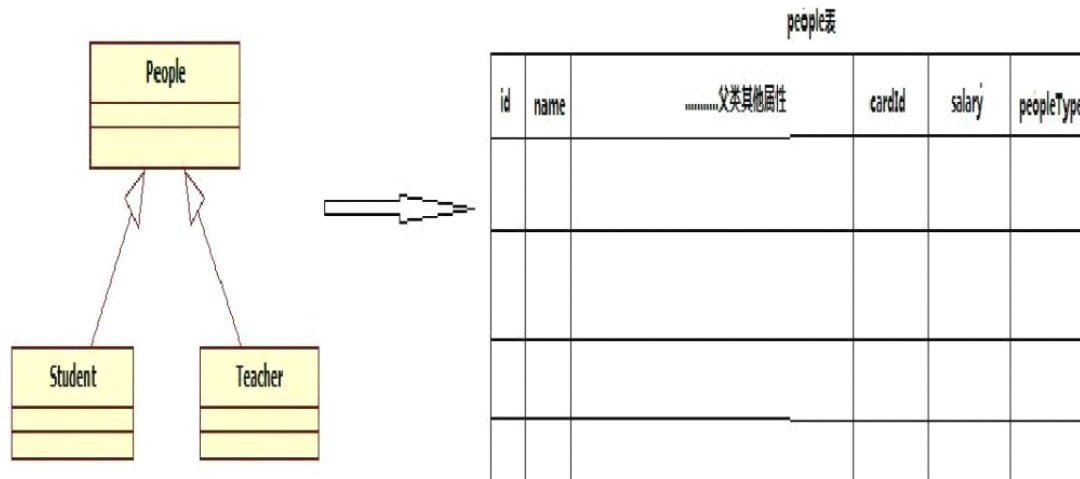
```

第二种方案：使用一张表表示所有继承体系下的类的属性的并集(TPH)

这种策略是使用<subclass>标签来实现的。因为类继承体系下会有许多个子类，要把多个类的信息存放在一张表中，必须有某种机制来区分哪些记录是属于哪个类的。**Hibernate** 中的这种机制就是，在表中添加一个字段，

用这个字段的值来进行区分。在表中添加这个标示列使用<discriminator>标签来实现。

该策略的示意图：



将继承体系中的所有类信息表示在同一张表中后，只要是这个类没有的属性会被自动赋上 null。

配置 People.hbm.xml:

[html] [view plaincopyprint?](#)

1. `<?xml version="1.0" encoding="utf-8"?>`
- 2.
3. `<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">`
- 4.
- 5.
- 6.
7. `<hibernate-mapping>`
- 8.
9. `<class name="com.bzu.hibernate.pojos.People" table="people">`
- 10.
11. `<id name="id" type="integer">`
- 12.

```

13.     <column name=      ></column>
14.
15.     <generator class=      ></generator>
16.
17. </id>
18.
19.
20.
21. <discriminator column="peopleType"      ="string"></discriminator>
22.
23.
24.
25. <property name=      column="name"      ="string"></property>
26.
27. <property name=      column="sex"      ="string"></property>
28.
29. <property name=      column="age"      ="string"></property>
30.
31. <property name=      column="birthday"      ="timestamp"></property>
32.
33.
34.
35. <subclass name="com.bzu.hibernate.pojos.Student"      -value=
    >
36.
37.     <property name=      column="cardId"      ="string"></property>
38.
39. </subclass>
40.
41.
42.
43. <subclass name="com.bzu.hibernate.pojos.Teacher"      -value=
    >
44.

```

```

45.     <property name=      column="salary"      ="string"></property>
46.
47.     </subclass>
48.
49. </class>
50.
51. </hibernate-mapping>

```

`<discriminator>`标签用于在表中创建一个标识列，其"column"属性指定标识列的列名，"type"指定了标识列的类型。`<subclass>`标签用于指定该 HBM 文件代表类的子类，有多少子类就有多少个该标签，其"name"属性指定子类的名称，"discriminator-value"属性指定该子类的数据的标识列的值是什么，其"extends"属性与`<union-subclass>`的"extends"属性用法一致。

下面来看一下根据 `People.hbm.xml` 生成表结构，可以看到一张表将继承体系下的所有信息都包含了，其中"peopleType"为标识列：

[sql] [view plaincopyprint?](#)

```

1. drop table if exists people
2.
3.   createtable people (
4.
5.       idvarchar(255) not null,
6.
7.       peopleType varchar(255) not null,
8.
9.       namevarchar(255),
10.
11.       sexvarchar(255),
12.
13.       agevarchar(255),

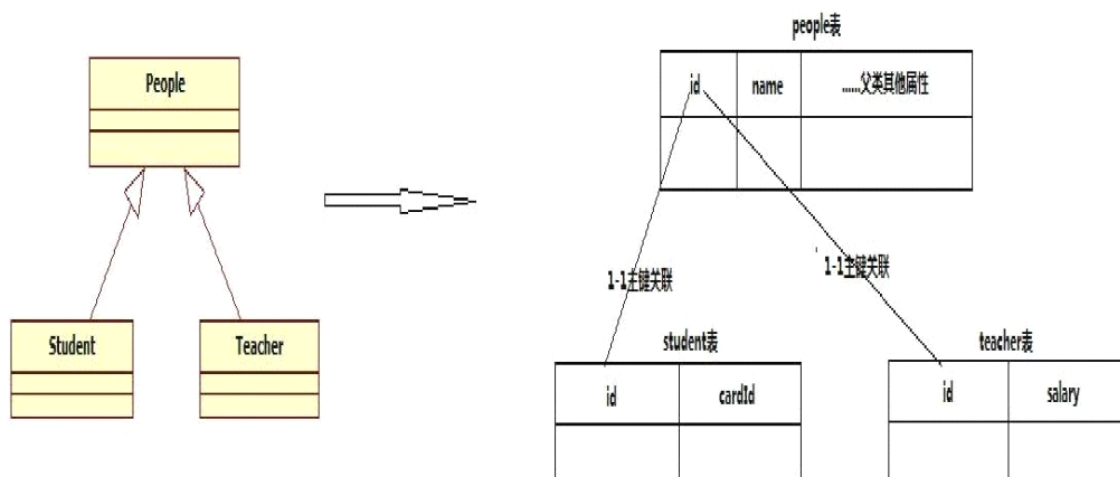
```

```
14.  
15.    birthday datetime,  
16.  
17.    cardIdvarchar(255),  
18.  
19.    salaryvarchar(255),  
20.  
21.    primary key (id)  
22.  
23. )
```

第三种方案：每个子类使用一张表只存储它特有的属性，然后与父类所对应的表以一对一主键关联的方式关联起来。（TPC）

这种策略是使用<joined-subclass>标签来定义子类的。父类、子类都对应一张数据库表。在父类对应的数据库表中，它存储了所有记录的公共信息，实际上该父类对应的表会包含所有的记录，包括父类和子类的记录；在子类对应的数据库表中，这个表只定义了子类中所特有的属性映射的字段。子类对应的数据表与父类对应的数据表，通过一对一主键关联的方式关联起来。

这种策略的示意图：



people 表中存储了子类的所有记录，但只记录了他们共有的信息，而他们独有的信息存储在他们对应的表中，一条记录要获得其独有的信息，要通过 **people** 记录的主键到其对应的子表中查找主键值一样的记录然后取出它独有的信息。

配置 **People.hbm.xml**:

[html] [view plaincopyprint?](#)

1. `<?xml version="1.0" encoding="utf-8"?>`
- 2.
3. `<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">`
- 4.
- 5.
- 6.
7. `<hibernate-mapping>`
- 8.
9. `<class name="com.suxiaolei.hibernate.pojos.People" table="people">`
- 10.
11. `<id name="id" type="integer">`
- 12.

```

13.     <column name=      ></column>
14.
15.     <generator class=      ></generator>
16.
17. </id>
18.
19.
20.
21. <property name=      column="name"      ="string"></property>
22.
23. <property name=      column="sex"      ="string"></property>
24.
25. <property name=      column="age"      ="string"></property>
26.
27. <property name=      column="birthday"      ="timestamp"></property>
28.
29.
30.
31. <joined-subclassname>joined-subclassname="com.suxiaolei.hibernate.pojos.
    Student"      ="student">
32.
33.     <key column=      ></key>
34.
35.     <property name=      column="cardId"      ="string"></property>
36.
37. </joined-subclass>
38.
39.
40.
41. <joined-subclassname>joined-subclassname="com.suxiaolei.hibernate.pojos.
    Teacher"      ="teacher">
42.
43.     <key column=      ></key>
44.

```

```

45.     <property name=         column="salary"         ="integer"></property>
46.
47.     </joined-subclass>
48.
49. </class>
50.
51. </hibernate-mapping>

```

<joined-subclass>标签需要包含一个 **key** 标签，这个标签指定了子类和父类之间是通过哪个字段来关联的。

根据 **People.hbm.xml** 生成表结构，可以看到，父类对应的表保存公有信息，子类对应的表保存独有信息，子类和父类对应的表使用一对一主键关联的方式关联起来

[sql] [view plaincopyprint?](#)

```

1. drop table if exists people
2.
3. drop table if exists student
4.
5. droptable if exists teacher
6.
7. create table people (
8.
9.     id varchar(255) not null,
10.
11.     name varchar(255),
12.
13.     sex varchar(255),
14.
15.     age varchar(255),

```



```

16.
17.     birthday datetime,
18.
19.     primary key (id)
20.
21. )
22.
23. create table student (
24.
25.     id varchar(255) not null,
26.
27.     cardId varchar(255),
28.
29.     primary key (id)
30.
31. )
32.
33. create table teacher (
34.
35.     id varchar(255) not null,
36.
37.     salary integer,
38.
39.     primary key (id)
40.
41. )
42.
43.
44.
45. alter table student
46.
47.     add index FK8FFE823BF9D436B1 (id),
48.
49.     add constraint FK8FFE823BF9D436B1
50.

```

51. foreign key (id)
 52.
 53. references people (id)
 54.
 55.
 56.
 57. alter table teacher
 58.
 59. add index FKAA31CBE2F9D436B1 (id),
 60.
 61. add constraint FKAA31CBE2F9D436B1
 62.
 63. foreign key (id)
 64.
 65. references people (id)

三种映射方式的比较和选择---三种方式的优缺点

为了方便说明为三种方式按顺序标号为[1][2][3]。

【1】： 优点：数据结构清晰

缺点：两个子表的主键不能重复，不能使用数据库的自增方式生成主键。

【2】： 优点：查询效率高，符合数据库设计粗粒度（推荐）

缺点：存在冗余字段，有些字段是子类不具有的属性。

【3】： 优点：数据结构清晰，没有冗余

缺点：类的继承层次比较多的话，造成生成的表也比较多，增删改查效率低

下

总结：

1. 通过总结这三种方式的优缺点发现，使用继承树生成一张表的方式似乎更符合数据库粗粒度设计的原则。当然数据量非常大的话也可以考虑每个类生成一张表的成方式

2. 程序的对象模型没有发生变化，变化的是关系模型。

这就是 **hibernate** 的好处，想改变关系模型，只需要改变映射文件即可。

（五十六）细谈 Hibernate（七）Hibernate 自身一对多和多对多关系映射

欢迎阅读本专题其他博客：

细谈 Hibernate（十）hibernate 查询排序和组件映射

细谈 Hibernate（十一）hibernate 复合主键映射

细谈 Hibernate（十二）hibernate 查询排序组件映射

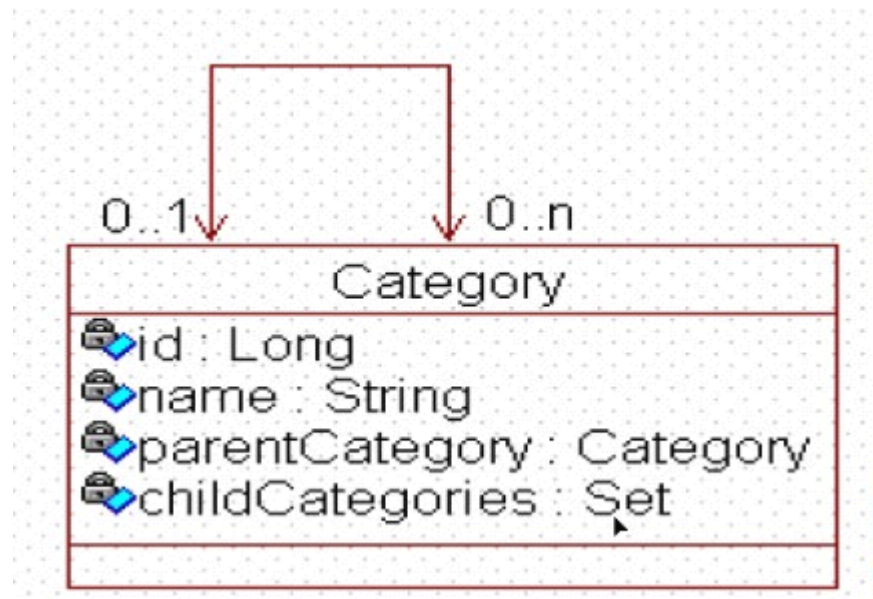
细谈 Hibernate（十三）session 缓存机制和三种对象状态

一对多关系映射大家都明白，关系双方都一个含有对方多个引用，但自身一对多很多同学都不明白什么意思，那么首先我就说明一下什么是自身一对多，其实也很好理解，自身一对多就是自身含有本身的多个引用，例如新闻类别，新闻包含体育新闻和政治新闻，体育新闻内有含有足球新闻和篮球

新闻，其实他们都属于新闻，只是名字不同而已，下面我们就以新闻类别为例来具体说明一下：

首先我们来看一下新闻类别的类图：

类图：category



从上面的图我们可以看出：每一个新闻类别都有一个父类别和一个孩子类别的 **set** 集合，这个父类别和孩子类别里面都是自身的引用，这样就行了自身一对多的对象关系

下面看一下具体的新闻实体类：Category.java

[java] [view plaincopyprint?](#)

```
1. public class Category
2.
3. {
4.
5.     private Long id;
6.
7.     private String name;
8.
```

```

9. private Category parentCategory;
10.
11.private Set<Category> childCategories;
12.
13.public Category(String name, Category parentCategory,
14.
15.Set<Category> childCategories)
16.
17.{
18.
19.this.name = name;
20.
21.this.parentCategory = parentCategory;
22.
23.this.childCategories = childCategories;
24.
25.}
26.
27.public Category()
28.
29.{
30.
31.}
32.
33.*****set、get 方法省略
34.
35.}

```

看完具体的实体类之后我们下面看一下其具体的配置，其实他的配置中没什么特别的地方，仅仅只是他的配置中包含了一对多和多对一的共同标签存在而已：他即含有多的一方的<set>标签。也含有一的一方的<many-to-one>标签：

Category.hbm.xml 配置文件

[html] view plaincopyprint?

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6.
7. <hibernate-mapping>
8.
9.   <class name="org.hibernate.example.User" table="user">
10.
11.     <id name="id" column="id" type="integer">
12.
13.       <generator class="org.hibernate.id.UUIDGenerator"></generator>
14.
15.     </id>
16.
17.     <property name="name" type="string">
18.
19.       <column name="name" length=255></column>
20.
21.     </property>
22.
23.     <set name="roles" cascade="all" inverse="false">
24.
25.       <key column="id"></key>
26.
27.       <one-to-many class="org.hibernate.example.Role"></one-to-many>
28.
29.     </set>
30.
31.     <many-to-one name="role" column="role_id" class="org.hibernate.example.Role">
32.
```

```
33. </many-to-one>
34.
35. </class>
36.
37. </hibernate-mapping>
```

下面我们来看一下在自身一对多的关系下进行增删改查的示例：

[java] [view plain](#)[copy](#)[print](#)?

```
1. import java.util.HashSet;
2.
3. import org.hibernate.Session;
4.
5. import org.hibernate.SessionFactory;
6.
7. import org.hibernate.Transaction;
8.
9. import org.hibernate.cfg.Configuration;
10.
11. public class HibernateTest2
12. {
13. {
14.
15. private static SessionFactory sessionFactory;
16.
17. static
18.
19. {
20.
21. try
22.
23. {
24.
25. sessionFactory = new Configuration().configure()
```



```
26.  
27..buildSessionFactory();  
28.  
29.}  
30.  
31.catch (Exception ex)  
32.  
33.{  
34.  
35.ex.printStackTrace();  
36.  
37.}  
38.  
39.}  
40.  
41.public static void main(String[] args)  
42.  
43.{  
44.  
45.Session session = sessionFactory.openSession();  
46.  
47.Transaction tx = null;  
48.  
49.try  
50.  
51.{  
52.  
53.tx = session.beginTransaction();  
54.  
55.Category category1 = new Category("level1", null, new HashSet<Category>());  
56.  
57.Category category2 = new Category("level2", null, new HashSet<Category>());  
58.  
59.Category category3 = new Category("level2", null, new HashSet<Category>());  
60.
```

```
61. Category category4 = new Category("level3", null, new HashSet<Category>());
62.
63. Category category5 = new Category("level3", null, new HashSet<Category>());
64.
65. Category category6 = new Category("level3", null, new HashSet<Category>());
66.
67. Category category7 = new Category("level3", null, new HashSet<Category>());
68.
69. category2.setParentCategory(category1);
70.
71. category3.setParentCategory(category1);
72.
73. category1.getChildCategories().add(category2);
74.
75. category1.getChildCategories().add(category3);
76.
77. category4.setParentCategory(category2);
78.
79. category5.setParentCategory(category2);
80.
81. category2.getChildCategories().add(category4);
82.
83. category2.getChildCategories().add(category5);
84.
85. category6.setParentCategory(category3);
86.
87. category7.setParentCategory(category3);
88.
89. category3.getChildCategories().add(category6);
90.
91. category3.getChildCategories().add(category7);
92.
93. Category category = (Category)session.get(Category.class, new Long(1));
94.
95. System.out.println(category.getChildCategories().iterator().next().getName());
```

```
96.  
97.session.delete(category);  
98.  
99.tx.commit();  
100.  
101.    }  
102.  
103.    catch(Exception ex)  
104.  
105.    {  
106.  
107.        if(null != tx)  
108.  
109.        {  
110.  
111.            tx.rollback();  
112.  
113.        }  
114.  
115.    }  
116.  
117.    finally  
118.  
119.    {  
120.  
121.        session.close();  
122.  
123.    }  
124.  
125.    }  
126.  
127.    }  
128.  
129.
```

在很多实际开发过程中，多对多的映射关系也是比较常见的，最为明显的例子就是我们常用的学生选课示例，一个学生可以选多门课，一门课也可以由多个学生去选，这样就形成了多对多的映射关系，现在我们就以学生选课的实例来看一看多对多关系映射。由于在多对多映射中，双向多对多用的比较多，并且单向多对多也比较简单，所以我们就以双向多对多进行讲解

我们先把必要的实体类和实体映射文件写好：

先简单看一下实体类：

student.java

[java] [view plain](#)[copy](#)[print?](#)

```
1. /** 学生实体类 */
2.
3. public class Student {
4.
5.     private Long id;           //对象标识符(OID)
6.
7.     private String name;       //姓名
8.
9.     private String grade;      //所在班级
10.
11. private Set<Course> courses;  //所有所选课程的集合
12.
13. public Student(){}           //无参数的构造方法
14.
15. *****set、get 方法省略
16.
17. }
```

Course.java

[java] [view plaincopyprint?](#)

```
1. /** 课程实体类 */
2.
3. public class Course {
4.
5.     private Long id;           //对象标识符(OID)
6.
7.     private String name;       //课程名
8.
9.     private Double creditHours; //课时数
10.
11. private Set<Student> students; //选择了这门课程的学生的集合
12.
13. public Course(){}             //无参数的构造方法
14.
15. *****set、get 方法省略
16.
17. }
```

下一步编写实体映射文件：

Student.hbm.xml

[html] [view plaincopyprint?](#)

```
1. <?xml version=      encoding=      ?>
2.
3. <!DOCTYPE hibernate-mapping PUBLIC
4.
5.     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
6.
7.     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
8.
9. <hibernate-mapping>
10.
```

```

11. <!-- 映射持久化类 -->
12.
13. <class name=          table=          >
14.
15. <!-- 映射对象标识符 -->
16.
17. <id name=      column=      type=      >
18.
19. <generator class=      />
20.
21. </id>
22.
23. <!-- 映射普通属性 -->
24.
25. <property name=      />
26.
27. <property name=      />
28.
29. <!-- 映射集合属性,指定连接表 -->
30.
31. <set name=          table=          >
32.
33. <!-- 用 key 元素指定本持久类在连接表中的外键字段名 -->
34.
35. <key column=      />
36.
37. <!-- 映射多对多关联类 -->
38.
39.      <many-to-many column=
40.
41.      class=          />
42.
43. </set>
44.
45. </class>

```

```

46.
47. </hibernate-mapping>
48.
49. Course.hbm.xml:
50.
51. <?xml version=      encoding=      ?>
52.
53. <!DOCTYPE hibernate-mapping PUBLIC
54.
55.     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
56.
57.     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
58.
59. <hibernate-mapping>
60.
61. <!-- 映射持久化类 -->
62.
63. <class name=      table=      >
64.
65. <!-- 映射对象标识符 -->
66.
67. <id name=      column=      type=      >
68.
69. <generator class=      />
70.
71. </id>
72.
73. <!-- 映射普通属性 -->
74.
75. <property name=      />
76.
77. <property name=      column=      />
78.
79. <!-- 映射集合属性,指定连接表 -->
80.

```

```

81. <set name=          table=          inverse=      >
82.
83. <!-- 用 key 元素指定本持久类在连接表中的外键字段名 -->
84.
85. <key column=          />
86.
87. <!-- 映射多对多关联类 -->
88.
89. <many-to-many column=          class=          />
90.
91. </set>
92.
93. </class>
94.
95. </hibernate-mapping>

```

下面具体看一下增删改查的具体测试的关键代码：

增加数据测试：

[java] [view plaincopyprint?](#)

```

1. <SPAN xmlns="http://www.w3.org/1999/xhtml"><SPAN xmlns="http://www.w3.org
   /1999/xhtml">tran = session.beginTransaction();
2.
3. Student stu1 = new Student("xiaoli", "two");
4.
5. Student stu2 = new Student("xiaoming", "two");
6.
7. Student stu3 = new Student("xiaoqiang", "two");
8.
9. Course course1 = new Course("java", 3.0);
10.
11. Course course2 = new Course("c#", 5.0);

```



```
12.  
13.//stuset.add(stu1);  
14.  
15.//stuset.add(stu2);  
16.  
17.//course1.setStudents(stuset);  
18.  
19.//session.save(course1);  
20.  
21.couset.add(course1);  
22.  
23.couset.add(course2);  
24.  
25.stu1.setCourses(couset);  
26.  
27.session.save(stu1);  
28.  
29.tran.commit();</SPAN></SPAN>
```

测试结论：如果想保存数据成功，不管是主控方还是被控方，如果想通过一次保存即可把双方数据保存，需要把实体配置中的 **cascade** 属性设置为 **all** 或者 **save-update**，由于设置为 **all** 包含 **delete**，在删除数据中，删除一条信息会导致相对应表的多条或者全部信息被删掉，所以一般配置 **save-update**。

（五十七）细谈 Hibernate （八）Hibernate 集合 Map 关系映射

对于 **hibernate** 中,集合属性在 **Hibernate** 的映射文件中是非常常见的,也是非常重要的内容,理解和熟练掌握常用的集合属性则显得更为重要。在 **hibernate** 的配置文件中,例如每个人的考试成绩,就是典型的 **Map** 结构,每门功课对应一门成绩。或者更简单的集合属性,某个企业的部门,一个企业通常对应多个部门等。集合属性是现实生活中非常普遍的属性关系。集合属性大致有两种:第一种是单纯的集合属性,例如像 **List**,**Set** 或数组等集合属性;还有一种就是 **Map** 结构的集合属性。每个属性都有对应的 **key** 映射。集合属性的元素大致有如下几种:

（1）**<set>**元素: 可以映射类型为 **java.util.Set** 接口的属性,它的元素存放没有顺序且不允许重复,也可以映射类型为 **java.util.SortSet** 接口的属性,它的元素可以按自然属性排序

（2）**<list>**元素: 可以映射类型为 **java.util.List** 接口的属性,它需要在结合属性对象的数据库表中用一个额外的索引列保存每一个元素的位置,即是有属性可重复的。

（3）**<bag>**元素: 可以映射 **java.util.Collection** 接口的属性,它的元素可能重复,但不保存属性,和 **set** 差不多,正因为有它,是因为如果通常使用 **list** 比较多,并且不想让添加一列的话,就用它。

(4) **<map>元素**: 可以映射为 `java.util.Map` 接口的属性, 它的元素以键值对的形式保存, 也是无序的, 也可以映射类型为 `java.util.SortMap` 接口的属性, 它的元素可以按自然顺序排序。

(5) **<array>元素**: 可以映射类型为数组的属性, 但在实际运用中用的极少

在这篇博客我们主要来看一下, `map` 属性映射的详细情况

`Map` 属性也是比较常见的属性类型, 比如, 一个组队内有多个学生, 学生就是一个名字对应着相应的名字, 下面我们就根据这个示例详细看一下:

首先看一下实体: `team.java`

[java] [view plaincopyprint?](#)

```
1. public class Team
2.
3. {
4.
5.     private Stringid;
6.
7.     private StringteamName;
8.
9.     private Mapstudents = new HashMap();
10.
11.     *****set、get 方法
12.
13. }
```

下面我们来具体看一下具体的实体映射配置

`Team.hbm.xml`

[html] view plaincopyprint?

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6.
7. <hibernate-mapping>
8.
9.   <class name="com.example.entity.Team" table="team">
10.
11.     <id name="id" column="id" type="integer">
12.
13.       <generator class="org.hibernate.id.UUIDGenerator"></generator>
14.
15.     </id>
16.
17.     <property name="name" column="teamName" type="string"></property>
18.
19.     <map name="members" table="team_members">
20.
21.       <key column="team_id"></key>
22.
23.       <index column="member_id" type="integer"></index>
24.       <!-- 指定的是 Map 中的 key 值 -->
25.
26.       <element column="member_name" type="string"></element>
27.       <!-- 指定的是 Map 中的 value 值 -->
28.
29.     </map>
30.   </class>
```

30.

31. `</hibernate-mapping>`

从上边配置文件可以看出，其实 `map` 属性映射的配置文件其他的配置都和我们以前配置一样，只是在配置 `map` 属性的时候有点不一样而已，下面我们就具体看一下 `map` 属性配置：首先要给大家说明的是，**虽然我们只有一个配置文件，但是上面的配置会生成两个数据库表的，`map` 属性里的会单独生成一个数据库表，这个表包含 `map` 的 `key` 和 `value`，还有一个外键。**`map` 属性配置内的 `key` 标签指定的是 `map` 对应表中数据所参考的 `team` 的 `id`，也就是说这个外键对应 `team` 的主键。`Map` 标签内的 `index` 标签对应指的是 `map` 数据中的 `key` 值，`element` 标签指定的是 `map` 数据中的 `value` 值。

其实上面我们看到的 `map` 中 `value` 值只是简单数据类型，但是在实际开发中，这里的 `value` 大多数都是对象的形式。现在我们还是以上面的例子来看一下，这里的 `value` 是对象的时候我们该如何配置：当 `value` 是对象的时候，我们就不能再简单的用一个配置文件来生成两个数据库表了，这时我们需要建立我们的 `team` 类。还需要建立 `value` 对应的 `student` 实体类

首先我们来看一下实体：Team.java

[java] [view plain](#)[copy](#)[print?](#)

```
1. public class Team
2.
3. {
4.
5.     private String id;
```

```
6.  
7.    private StringteamName;  
8.  
9.    private Mapstudents = new HashMap();  
10.  
11.*****省略 set、get 方法}
```

Student.java

[java] [view plaincopyprint?](#)

```
1. public class Student  
2.  
3. {  
4.  
5.    private String id;  
6.  
7.    private StringcardId;  
8.  
9.    private Stringname;  
10.  
11.   private int age;  
12.  
13.   private Team team;  
14.  
15.*****省略 set、get 方法  
16.  
17.}
```

下面我们继续来看一下相关的实体映射文件

Team.hbm.xml:

[html] [view plaincopyprint?](#)

```

1. <?xml version="1.0" encoding="UTF-8" ?>
2.
3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6.
7.
8.
9. <hibernate-mapping>
10.
11.
12.
13. <class name="Team" table="teams">
14.
15.
16.
17. <id name="id" column="id" type="integer">
18.
19. <generator class="org.hibernate.id.IntegerGenerator"></generator>
20.
21. </id>
22.
23.
24.
25. <property name="teamName" column="teamName" type="string"></property>
26.
27.
28.
29. <map name="teamMembers" table="teamMembers" cascade="all">
30.
31. <key column="teamId"></key>
32.

```

```

33.         <index column="card_id" type="java.lang.String"></index> <!--指定
    的是 Map 中的 key 值 -->
34.
35.         <one-to-manyclass one-to-manyclass="
    />
36.
37.     </map>
38.
39. </class>
40.
41. </hibernate-mapping>

```

通过上面的配置文件我们可以看出，这个配置文件几乎和简单数据类型的配置文件差不多，仅仅不同的是 **map** 标签内的 **element** 标签换成了 **one-to-many** 标签，这样充分可以说明：

map 标签中的 **element** 子标签映射的是原子类型（string, date, int, long...），即能够直接映射到数据库表字段上的类型，而 **one-to-many** 映射的则是实体类型，指的是无法映射到表的某个字段，而是要映射到整张表的类型。**但有一点需要注意的是，在 **map** 标签中的 **one-to-many** 中不要设置 **inverse=true**，因为如果要想让 **map** 值中的对象去维护两者之间的关系的话，在保存数据中很有可能外键添加不上，被设置为 **null****

下面在来看一下 student.hbm.xml

[\[html\] view plaincopyprint?](#)

```

1. <?xml version="1.0" encoding="UTF-8"?>
2.

```


3. <!DOCTYPEhibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

4.

5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

6.

7. <hibernate-mapping>

8.

9. <classnameclassname="com.shengsiyuan.hibernate.Student" = "student">

10.

11. <id name= column="id" = "string">

12.

13. <generatorclassgeneratorclass= ></generator>

14.

15. </id>

16.

17.

18.

19. <property name="cardId" = "card_id" type= ></property>

20.

21. <property name="name" = "name" type= ></property>

22.

23. <property name="age" = "age" type= ></property>

24.

25. <many-to-one name="team" = "team_id" class="com.shengsiyuan.hibernate.Team" = "none" fetch= >

26.

27. </many-to-one>

28.

29. </class>

30.

31. </hibernate-mapping>

细心的同学可以发现，这个配置文件和一对多关系映射中的多的一方配置的差不多，其实就是一样的，所以在这里就不赘述了。

（五十八）细谈 Hibernate（九）hibernate 一对一关系映射

一对一关系映射即为关系双方都含有对方一个引用，其实在生活中一对一关系也很常见，比如人和身份证，学生和学号等，都是一对一的关系映射，一对一映射分为单向的和双向的，没种关系映射又可以分为主键关联映射，唯一外键关联映射。

一：主键关联映射

一般一对一主键关联映射通过 **foreign** 主键生成器使用另外一个相关联的对象的标识符。通常和<one-to-one>联合起来使用。一对一主键关联映射原理：让两个实体的主键一样，这样就不需要加入多余的字段。**此种关联映射有一定的缺点：单向一对一主键关联实际上限制很多,因为你只有 IdCard 插入了那才能有这个 Person.**我们看一下具体示例：



根据上面的关系类图，我们再来看一下实体类的定义

实体 IdCard.java

[java] [view plaincopyprint?](#)

```
1. public class IdCard
2.
3. {
4.
5.     private String id;
6.
7.     private int number;
8.
9.     private Person person;
10.
11. }
```

实体: Person.java

[java] [view plaincopyprint?](#)

```
1. public class Person
2.
3. {
4.
5.     private String id;
6.
7.     private String name;
8.
9.     private IdCard idCard;
10.
11. }
```

IdCard 的配置文件:

[\[html\] view plaincopyprint?](#)

1. `<?xml version= ?>`
2. `<!DOCTYPE hibernate-mapping PUBLIC`
3. `"-//Hibernate/Hibernate Mapping DTD3.0//EN"`
4. `"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">`
5. `<hibernate-mapping>`
6. `<classnameclassname= table= >`
- 7.
8. `<!--这里我们应该注意的是此处的主键生成策略，这里的主键是利用的外键生成策略生成的，这里的主键关联着 idcard 表中的主键，也就是说，保证 Person 和 idcard 的主键相同。`
- 9.
10. `-->`
11. `<id name= >`
12. `<generator class= />`
13. `</id>`
14. `<property name= />`
15. `</class>`
16. `</hibernate-mapping>`

person 的配置文件:

[\[html\] view plaincopyprint?](#)

1. `<?xml version= ?>`
2. `<!DOCTYPE hibernate-mapping PUBLIC`
3. `"-//Hibernate/Hibernate Mapping DTD3.0//EN"`
4. `"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">`
5. `<hibernate-mapping>`
6. `<classnameclassname= table= >`
7. `<!--id 引用外部主键 -->`
8. `<id name= >`

```

9.     <generatorclassgeneratorclass=      >
10.     <paramnameparamname=      >idCard</param>
11.     </generator>
12. </id>
13. <property name=      />
14. <one-to-one name="idCard"      ="true"/>
15. </class>
16. </hibernate-mapping>

```

注：one-to-one 标签告诉 hibernate 根据主键加载引用对象，把 person 中的主键拿到 idCard 表中进行查找，然后把查到的信息加载到引用对象中采用一对一主键约束，那么必须设置 **constrained** 属性，表示当前主键作为外键参照了该属性在一对一主键关联映射中默认问级联属性

配置完了以后我们来看一下具体的增删改查操作：

[java] [view plaincopyprint?](#)

```

1. Person person = newPerson();
2.
3.     person.setName("zhangsan");
4.
5.     IdCard idCard = new IdCard();
6.
7.     idCard.setNumber(987654);
8.
9.     person.setIdCard(idCard);
10.
11.     idCard.setPerson(person);
12.
13.     Session session = sessionFactory.openSession();
14.

```

```

15.         Transaction tx = null;
16.
17.
18.
19.         try
20.
21.         {
22.             tx =session.beginTransaction();
23.             session.save(person);
24.             tx.commit();
25.         }
26.         catch(Exception ex)
27.         {
28.             ex.printStackTrace();
29.             if(null != tx)
30.             {
31.                 tx.rollback();
32.
33.             }
34.         }
35.         finally
36.         {
37.             session.close();
38.         }
39.
40.         //-----
41.
42. //         Session session = sessionFactory.openSession();
43. //         Transaction tx = null;
44. //         Person person = null;
45. //         try
46. //         {
47. //             tx =session.beginTransaction();
48. //             person =(Person)session.get(Person.class,"402881ec2ebd7e77012eb
d7e79e40001");

```

```

49.//         tx.commit();
50.//     }
51.//     catch(Exception ex)
52.//     {
53.//         if(null != tx)
54.//         {
55.//             tx.rollback();
56.//         }
57.//     }
58.//     finally
59.//     {
60.//         session.close();
61.//     }
62.//     System.out.println(person.getName());
63.//     System.out.println(person.getIdCard().getNumber());
64.
65.//-----
66.
67.//     Session session = sessionFactory.openSession();
68.//     Transaction tx = null;
69.//
70.//     Person person = null;
71.//     try
72.//     {
73.//         tx =session.beginTransaction();
74.//
75.//         person =(Person)session.get(Person.class,"402881ec2ebd7e77012eb
            d7e79e40001");
76.//
77.//         person.setName("lisi");
78.//         tx.commit();
79.//     }
80.//     catch(Exception ex)
81.//     {
82.//         if(null != tx)

```



```

83.//      {
84.//          tx.rollback();
85.//      }
86.//  }
87.//  finally
88.//  {
89.//      session.close();
90.//  }
91.//  System.out.println(person.getName());
92.    //-----
93.
94.//  Session session = sessionFactory.openSession();
95.//  Transaction tx = null;
96.//  Person person = null;
97.//  try
98.//  {
99.//      tx =session.beginTransaction();
100.    //      person =(Person)session.get(Person.class,"402881ec2ebd7e7701
        2ebd7e79e40001");
101.    //      session.delete(person);
102.    //      tx.commit();
103.    //  }
104.    //  catch(Exception ex)
105.    //  {
106.    //      if(null != tx)
107.    //      {
108.    //          tx.rollback();
109.    //      }
110.    //  }
111.    //  finally
112.    //  {
113.    //      session.close();
114.    //  }

```

通过执行查询，我们可以发现，hibernate 的一对一默认执行的检索方式是外连接检索方式，如果我们不想用外连接检索方式，我们可以设置一下 one-to-one 的 fetch 属性，他有两个值，一个是 select，一个是 join。我想大家通过字面也能猜出他们所对应的检索方式。

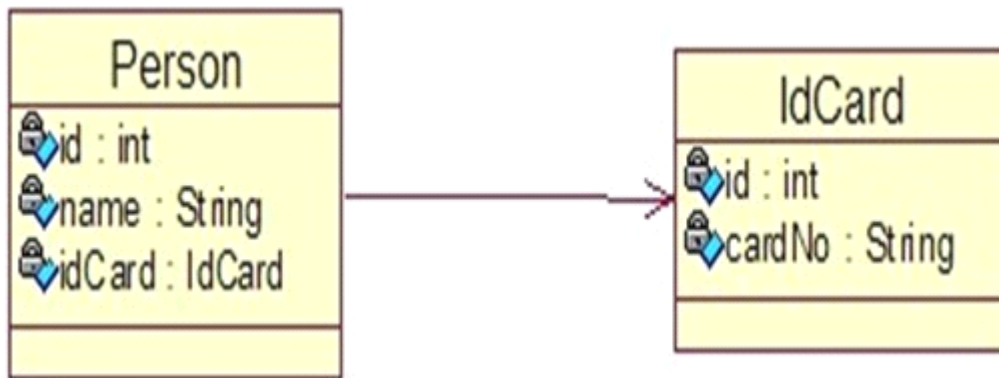
Hibernate 一对一中也可以设置延迟加载，一对一默认使用的是立即加载，如果需要使用延迟加载，那么需要在 one-to-one 元素中将 constrained 属性设为 true，并且将待加载的一方的 class 元素中的 lazy 属性设为 true（或者不去设置，因为该属性默认值就是 true）。一对一加载时默认使用左外连接，可以通过修改 fetch 属性为 select 修改成每次发送一条 select 语句的形式。

唯一外键关联映射：其实它是一对多的特殊情况，它基本和一对多是完全相同的，只不过需要配置一个属性而已。其本质上是一对多的蜕化形式。在 many-to-one 元素中增加 unique="true" 属性就变成了一对一。

二、一对唯一外键关联映射——单向

1. 一对唯一外键关联映射是多对一关联映射的特例，可以采用 <many-to-one> 标签，指定多的一端的 unique=true，这样就限制了多的一端的多重性为一，通过这种手段映射一对一唯一外键关联

2. 领域模型图：



3.配置

Person.hbm.xml:

[\[html\] view plaincopyprint?](#)

```

1. <class name="com.bjsxt.hibernate.Person"      ="t_person">
2.
3. <id name=      >
4.
5. <generator class=      />
6.
7. </id>
8.
9. <property name=      />
10.
11. <many-to-one name=      unique=      />
12.
13. </class>
  
```

IdCard.hbm.xml:

[\[html\] view plaincopyprint?](#)

```

1. <class name="com.bjsxt.hibernate.IdCard"      ="t_idcard">
2.
3. <id name=      >
4.
5. <generator class=      />
  
```

```

6.
7. </id>
8.
9. <property name=           />
10.
11.</class>

```

三、 一对唯一外键关联映射——双向

1.一对一唯一外键关联双向，需要在另一端（idcard），添加<one-to-one>标签，指示 hibernate 如何加载其关联对象，默认根据主键加载 person，外键关联映射中，因为两个实体采用的是 person 的外键维护的关系，所以不能指定主键加载 person，而要根据 person 的外键加载，所以采用如下映射方式：

```
<one-to-one name="person"property-ref="idCard"/>
```

2.领域模型图：



3.具体配置：

Person.hbm.xml:

[\[html\] view plaincopyprint?](#)

```

1. <class name="com.bjsxt.hibernate.Person"           ="t_person">
2.
3. <id name=      >

```

```

4.
5. <generator class=      />
6.
7. </id>
8.
9. <property name=      />
10.
11. <many-to-one name=      unique=      />
12.
13. </class>

```

IDCard.hbm.xml

[html] [view plain](#) [copy print?](#)

```

1. <class name="com.bjsxt.hibernate.IdCard"      ="t_idcard">
2.
3. <id name=      >
4.
5. <generator class=      />
6.
7. </id>
8.
9. <property name=      />
10.
11. <one-to-one name="person"      -ref=      />
12.
13. </class>

```

（五十九）细谈 Hibernate（十）hibernate 查询排序和组件映射

在实际开发过程中,有很多用户需要时要把查询出来的结果进行排序显示,而不是在数据库里面那样顺序混乱那样的显示,这样的话我们不得不对数据进行排序了, **hibernate** 对数据排序提供了很好的支持, **hibernate** 提供了两种对查询到得数据结果进行排序: **1**: 数据库排序,也就是说在数据库内部就进行完了排序。**2**.内存排序,也就是说在数据库中把数据加载到内存中在进行排序。其实一般我们推荐使用第二种排序方式,因为在数据库中排序的性能要远远高于在内存中排序的性能。

一：数据库排序

数据库排序主要是使用集合标签中的 **order-by** 属性,格式主要是为:
order-by="字段名 排序方式"; 例如: **order-by="name asc"**name 是指数据

库字段 `asc` 是升序，在 `hibernate` 中，`<set>`、`<idbag>`、`<map>`、`<list>` 元素都有 `order-by` 属性，如果设置了该属性，`Hibernate` 会利用 `order by` 子句进行排序，使用 `order-by` 属性，我们可以通过 `hbm` 文件执行生成的 `SQL` 如何使用 `orderby` 查询子句以返回排序后的结果集。下面我们就以一个具体的实例来具体看一下数据库排序的内容

我们就以学生和团队的关系来说一下：首先来看一下实体之间的数据结构关系：

Student.java

[java] [view plaincopyprint?](#)

```
1. public class Student {
2.
3.     private String id;
4.
5.     private String name;
6.
7.     private String description;
8.
9.     private Team team;
10.
11. ....*****set、get 方法省略
12.
13. }
```

Team.java

[java] [view plaincopyprint?](#)

```
1. public class Team {
2.
3.     private String id;
```

```

4.
5.   private String teamname;
6.
7.   private Set students;
8.
9.   *****set、get 方法省略
10.
11.}

```

从实体上我们可以看出，学生和团队是一个多对一得数据关系，这个我们以前都看过，也写过，相信大家都有已经很熟悉了。所以在此具体的配置文件我们也不多写了，我们主要来看一下配置排序的地方，下面我们看一下配置

[html] [view plaincopyprint?](#)

```

1. <!-- 以名称降序返回 student 集合 -->
2.
3.   <set name=           table=           cascade=   order-by=
4.   >
5.   <key column=         ></key>
6.
7.   <one-to-many class=           />
8.
9. </set>

```

从上面可以看出，其实配置数据库排序很简单，仅仅是在集合标签上配置一个 **order-by** 属性即可。

下面我们就具体来看一下测试代码：

[java] [view plaincopyprint?](#)

```
1. Transaction t=session.beginTransaction();
2.
3.
4.
5.     Team team=(Team)session.createQuery("from Team t where t.teamname='team1'").uniqueResult();
6.
7.     Set result=team.getStudents();
8.
9.     for (Iterator iterator = result.iterator(); iterator.hasNext();) {
10.
11.         Student object = (Student) iterator.next();
12.
13.         System.out.println(object.getName());
14.
15.     }
16.     t.commit();
```

运行这块代码，我们一起来看一下控制台的打印结果：

测试结果：

[html] [view plaincopyprint?](#)

```
1.     Hibernate: select team0_.id as id1_, team0_.teamname
as teamname1_ from teamOrder team0_ where team0_.teamname
=
2.     Hibernate: select students0_.team_id as team4_1_, students0_.id as id1_, students0_.id as id0_0_, students0_.name as name0_0_, students0_.description as descript3_0_0_, students0_.team_id as team4_0_0_ from studentOrder stude
```

```
nts0_ where students0_.team_id=? order by students0_.name
desc
3.      hello
4.      default
5.      bug<SPAN style=
```

>

从上面输出的 sql 语句就可以看出，我们查询到得数据时以 student 表中的 name 进行排序的。所以我们数据库排序的配置就这么结束了。

二．内存排序

内存排序，顾名思义，就是在内存中排序，把查询到得结果加载到内存以后惊醒排序。**Hibernate** 在配置文件中也给我提供了内存排序的配置，那就是 **sort** 属性，它有两个属性值可以直接使用，分别是 **unsorted**（不排序）以及 **natural**（自然排序，即升序），此外，我们还可以自定义排序规则，方式是定义一个类，让其实现 **Comparator** 接口，并且实现该接口中的 **compare** 方法，在该方法中实现排序规则即可。然后将该自定义排序规则类名作为 **sort** 的属性值即可。**<set>**和**<map>**元素都具有 **sort** 属性，如果设置了该属性，就会对内存中的集合对象进行排序。

<set>元素的 **sort** 属性为 **natural**，表示对集合中的字符串进行自然排序。

Hibernate 采用 **org.hibernate.PersistentSortedSet** 作为 **Set** 的实现类，**PersistentSortedSet** 类实现了 **java.util.SortedSet** 接口。当 **Session** 保存一个对象时，会调用 **org.hibernate.type.SortedSetType** 类的 **wrap()**方法，把

对象的集合属性包装为 **SortedSet** 类的实例，下面我们看一下 **wrap()**方法的源代码如下：

[java] [view plaincopyprint?](#)

```
1. public PersistentCollection wrap
2. (SessionImplementor session, Object collection) {
3.
4.     return new PersistentSortedSet
5. ( session, (java.util.SortedSet) collection );
6.
7. }
```

从 **wrap()**方法的源代码看出，应用程序中创建的对象集合属性必须是 **java.util.SortedSet** 类型，否则以上 **wrap()**方法会抛出 **ClassCastException**。其实内存排序和数据库排序是一样的，只是配置的参数不同而已，都是在集合标签配置一下，所以在此我们就不以示例演示了。

三、组件映射 (**component**)

在 **hibernate** 中，**component** 是某个实体对象的逻辑组成部分，它与实体的根本区别是，**component** 是没有标识的，它是一个逻辑组成部分，完全从属于某个实体，这样就在传统数据库上，实现了对象的细粒度划分，层次分明，实现了面向对象的领域划分
见下图：



把 **User** 和 **Employee** 中共同的属性（各种联系方式）拿出来，放在（抽象到）到一个单独的类中。这样物理上看有三个类，不过实体类还是只有 **User** 和 **Employee** 两个，也就是说数据库里只有 **User** 和 **Employee** 两张表。

下面我们就来看一下 **user** 和 **contact** 的实体及相关配置：

user.java

[java] [view plaincopyprint?](#)

```

1. public class User {
2.
3.     private int id;
4.
5.     private String name;
6.
7.     //联系方式
8.
9.     private Contact contact;
10.
11.     *****set、get 省略
12.
13. }
  
```

contact.java

[java] [view plaincopyprint?](#)

```

1. public class Contact {
  
```

```

2.
3.     privateString address;
4.
5.     privateString contactTel;
6.
7.     privateString email;
8.
9.     privateString zipCode;
10.
11. *****set、get 省略
12.
13.}

```

user.hbm.xml

[html] [view plaincopyprint?](#)

```

1. <?xmlversionxmlversion=    ?>
2.
3. <!DOCTYPEhibernate-mapping PUBLIC
4.
5.     "-//Hibernate/Hibernate MappingDTD 3.0//EN"
6.
7.     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
8.
9. <hibernate-mapping>
10.
11.     <classnameclassname=                table=                >
12.
13.         <idnameidname=    >
14.
15.             <generatorclassgeneratorclass=    />
16.
17.         </id>

```

```

18.
19.         <propertynamepropertyname=        />
20.
21.         <componentnamecomponentname=        >
22.
23.             <propertynamepropertyname=        />
24.
25.             <propertynamepropertyname=        />
26.
27.             <propertynamepropertyname=        />
28.
29.             <propertynamepropertyname=        />
30.
31.         </component>
32.
33.     </class>
34.
35.</hibernate-mapping>

```

从配置上来看，其实这个地方很好理解，**user** 把 **contact** 看做是组成的一部分，只是把他抽出一个单独的实体类了而已，这也正好体现了代码的复用性，其实他就是一对一关系的映射。

如果想要生成两张表，**hibernate** 也提供了相关的配置机制，其实只是换了换标签而已，把 **component** 标签换成 **composite-element**，仅此而已，这样就能生成两张表了。

（六十）细谈 Hibernate（十一）hibernate 复合主键映射

所谓复合主键就是在一张数据库表中，主键有两个或者多个，在日常开发中会遇到这样一种情况，数据库中的某张表需要多个字段列才能唯一确定一行记录，这时表需要使用复合主键。这是我们以前在 **hibernate** 配置中没有遇到过的情况。面对这样的情况 **Hibernate** 为我们提供了两种方式来解决复合主键问题，下面让我们来看一下这两种情况：

- 1：将复合主键对应的属性与实体其他普通属性放在一起
- 2：将主键属性提取到一个主键类中，实体类只需包含主键类的一个引用

下面我们就具体来看一下：

方式一：将复合主键对应的属性与实体其他普通属性放在一起

例如实体类 **People** 中" id "和" name "属性对应复合主键：

[java] [view plaincopyprint?](#)

```
1. /*实体类，使用复合主键必须实现 Serializable 接口*/
2. public class People implements Serializable
3. {
4.     private String id;
5.     private String name;
6.     private int age;
7.
8.     public People()
9.     {}
10. *****set、get 方法
11.
12.     public int hashCode()
13.     {
14.         final int prime = 31;
15.         int result = 1;
16.         result = prime * result + ((id == null) ? 0 : id.hashCode());
17.         result = prime * result + ((name == null) ? 0 : name.hashCode());
18.         return result;
19.     }
20.
21.     @Override
22.     public boolean equals(Object obj)
23.     {
24.         if (this == obj)
25.             return true;
26.         if (obj == null)
```



```

27.         return false;
28.     if (getClass() != obj.getClass())
29.         return false;
30.     People other = (People) obj;
31.     if (id == null)
32.     {
33.         if (other.id != null)
34.             return false;
35.     }
36.     else if (!id.equals(other.id))
37.         return false;
38.     if (name == null)
39.     {
40.         if (other.name != null)
41.             return false;
42.     }
43.     else if (!name.equals(other.name))
44.         return false;
45.     return true;
46. }
47.}

```

People.hbm.xml:

[html] [view plain](#)[copy](#)[print](#)?

```

1. <?xml version=      encoding=      ?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.
   0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
3.
4. <hibernate-mapping>
5.     <class name=      table=      >
6.         <!-- 复合主键使用 composite-id 标签 -->
7.         <composite-id>
8.             <!-- key-property 标签表示哪一些属性对应复合主键 -->

```

```

9.      <key-property name=      column=      type=      ></key-property>
10.     <key-property name=      column=      type=      ></key-proper
      ty>
11.     </composite-id>
12.
13.     <property name=      column=      type=      ></property>
14. </class>
15.</hibernate-mapping>

```

Hibernate 中使用复合主键时需要注意一些规则：

1. 使用复合主键的实体类必须实现 **Serializable** 接口。必须实现 **Serializable** 接口的原因很简单，我们查找数据的时候是根据主键查找的。打开 **Hibernate** 的帮助文档我们可以找到 **get** 与 **load** 方法的声明形式如下：

[java] [view plaincopyprint?](#)

```

1. Object load(Class theClass,Serializable id)
2.
3. Object get(Class theClass,Serializable id)

```

当 我们查找复合主键类的对象时，需要传递主键值给 **get()**或 **load()**方法的 **id** 参数，而 **id** 参数只能接收一个实现了 **Serializable** 接口的对 象。而复合主键类的主键不是一个属性可以表示的，所以只能先 **new** 出复合主键类的实例（例如：**new People()**），然后使用主键属性的 **set** 方法将主键值赋值给主键属性，然后将整个对象传递给 **get()**或 **load()**方法的 **id** 参数，实现主键值的 传递，所以复合主键的实体类必须实现 **Serializable** 接口。

2. 使 用复合主键的实体类必须重写 **equals** 和 **hashCode** 方法。必须重写 **equals** 和 **hashCode** 方法也很好理解。这两个方法使用于判断两个对象（两条记录）是否相等的。为什么要判断两个对象是否相等呢？因为数据库中的任意两条记录中的主键值是不能相同的，所以我们在程序中只要确保了两个对

象的主 键值不同就可以防止主键约束违例的错误出现。也许这里你会奇怪为什么不使用复合主键的实体类不重写这两个方法也没有主键违例的情况出现，这是因为使用单一 主键方式，主键值是 **Hibernate** 来维护的，它会确保主键不会重复，而复合主键方式，主键值是编程人员自己维护的，所以必须重写 **equals** 和 **hashCode** 方法用于判断两个对象的主键是否相同。

3. 重写的 **equals** 和 **hashCode** 方法，只与主键属性有关，普通属性不要影响这两个方法进行判断。这个原因很简单，主键才能决定一条记录，其他属性不能决定一条记录。

保存测试：

[java] [view plaincopyprint?](#)

```
1. tx = session.beginTransaction();
2.
3.     People people = new People();
4.     /*主键值由我们自己维护*/
5.     people.setId("123456");
6.     people.setName("zhangsan");
7.     people.setAge(40);
8.     session.save(people);
9.
10.tx.commit();
```

看看数据库：

	id *	name *	age
▶	123456	zhangsan	40

数据被正确的插入到数据库中了。

读取数据测试：

[java] view plaincopyprint?

```

1. tx = session.beginTransaction();
2.      /*查询复合主键对象，需要先构建主键*/
3.      People peoplePrimaryKey = new People();
4.      peoplePrimaryKey.setId("123456");
5.      peoplePrimaryKey.setName("zhangsan");
6.
7.      /*然后将构建的主键值传入 get 方法中获取对应的 People 对象*/
8.      People people = (People)session.get(People.class, peoplePrimaryKey);
9.      System.out.println("people age is:"+people.getAge());
10.     tx.commit();

```

控制台输出： people age is:40， 以看到数据成功的取出了。

方式二：将主键属性提取到一个主键类中，实体类只需包含主键类的一个引用。

主键类：

[java] view plaincopyprint?

```

1. /*必须实现 Serializable 接口*/
2. public class PeoplePrimaryKey implements Serializable
3. {
4.     /*复合主键值*/
5.     private String id;

```

```

6.    private String name;
7.
8.    public PeoplePrimaryKey()
9.    {}
10.
11.    /*复合主键值的 get 和 set 方法在此省略*/
12.
13.
14.
15.    @Override
16.    public int hashCode()
17.    {
18.        final int prime = 31;
19.        int result = 1;
20.        result = prime * result + ((id == null) ? 0 : id.hashCode());
21.        result = prime * result + ((name == null) ? 0 : name.hashCode());
22.        return result;
23.    }
24.
25.    @Override
26.    public boolean equals(Object obj)
27.    {
28.        if (this == obj)
29.            return true;
30.        if (obj == null)
31.            return false;
32.        if (getClass() != obj.getClass())
33.            return false;
34.        PeoplePrimaryKey other = (PeoplePrimaryKey) obj;
35.        if (id == null)
36.        {
37.            if (other.id != null)
38.                return false;
39.        }
40.        else if (!id.equals(other.id))

```

```

41.         return false;
42.     if (name == null)
43.     {
44.         if (other.name != null)
45.             return false;
46.     }
47.     else if (!name.equals(other.name))
48.         return false;
49.     return true;
50. }
51.}

```

实体类:

[java] [view plaincopyprint?](#)

```

1. public class People
2. {
3.     /*持有主键类的一个引用，使用该引用作为这个类的 OID*/
4.     private PeoplePrimaryKey peoplePrimaryKey;
5.     private int age;
6.
7.     public People()
8.     {}
9.     *****set/get 方法省略
10.
11.}

```

People.hbm.xml 文件稍有一点变动:

[html] [view plaincopyprint?](#)

```

1. <?xml version=      encoding=      ?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.
   0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

```

```

3.
4. <hibernate-mapping>
5.   <class name=                table=                >
6.     <!-- 复合主键使用 composite-id 标签 -->
7.     <!--
8.       name - 指定了复合主键对应哪一个属性
9.       class - 指定了复合主键属性的类型
10.    -->
11.     <composite-id name=                class=
12.                    >
13.       <!-- key-property 指定了复合主键由哪些属性组成 -->
14.       <key-property name=    column=    type=    ></key-property>
15.       <key-property name=    column=    type=    ></key-property>
16.     </composite-id>
17.     <property name=    column=    type=    ></property>
18.   </class>
19.</hibernate-mapping>

```

场景测试与方式一大同小异这里不再举例了。主键类为什么实现 `Serializable` 接口和为什么重写 `equals` 和 `hashCode` 方法上面已经解释的很清楚了。

3.联合主键的映射规则

1) 类中的每个主键属性都对应到数据表中的每个主键列。Hibernate 要求具有联合主键的实体类实现 `Serializable` 接口，并且重写 `hashCode` 与 `equals` 方法，重写这两个方法的原因在于 Hibernate 要根据数据库的联合主键来判断某两行记录是否是一样的，如果一样那么就认为是同一个对象，如果不一样，那么就认为是不同的对象。这反映到程序领域中就是根据 `hashCode` 与 `equals` 方法来判断某两个对象是否能够放到诸如 `Set` 这样的集合当中。联合主键的

实体类实现 **Serializable** 接口的原因在于使用 **get** 或 **load** 方法的时候需要先构建出来该实体的对象，并且将查询依据（联合主键）设置进去，然后作为 **get** 或 **load** 方法的第二个参数传进去即可。

2) 将主键所对应属性提取出一个类（称之为主键类），并且主键类需要实现 **Serializable** 接口，重写 **equals** 方法与 **hashCode** 方法，原因与上面一样。

（六十一）细谈 Hibernate（十二）hibernate 查询排序组件映射

在实际开发过程中，有很多用户需要时要把查询出来的结果进行排序显示，而不是在数据库里面那样顺序混乱那样的显示，这样的话我们不得不对数据进行排序了，**hibernate** 对数据排序提供了很好的支持，**hibernate** 提供了两种对查询到得数据结果进行排序：**1**：数据库排序，也就是说在数据库内部就进行完了排序。**2**.内存排序，也就是说在数据库中把数据加载到内存中在进行排序。其实一般我们推荐使用第二种排序方式，因为在数据库中排序的性能要远远高于在内存中排序的性能。

一：数据库排序

数据库排序主要是使用集合标签中的 **order-by** 属性，格式主要是为：**order-by="字段名 排序方式"**；例如：**order-by="name asc"**name 是指数据库字段 **asc** 是升序，在 **hibernate** 中，**<set>**、**<idbag>**、**<map>**、**<list>** 元素都有 **order-by** 属性，如果设置了该属性，**Hibernate** 会利用 **order by** 子句进行排序,使用 **order-by** 属性,我们可以通过 **hbm** 文件执行生成的 **SQL** 如何使用 **order by** 查询子句以返回排序后的结果集。下面我们就以一个具体的实例来具体看一下数据库排序的内容

我们就以学生和团队的关系来说一下:首先来看一下实体之间的数据结构关系:

Student.java

[java] [view plaincopyprint?](#)

```
1. public class Student {
2.     private String id;
3.     private String name;
4.     private String description;
5.     private Team team;
6.     .*****set、get 方法省略
7. }
```

Team.java

[java] [view plaincopyprint?](#)

```
1. public class Team {
2.     private String id;
3.     private String teamname;
4.     private Set students;
```

5. *****set、get 方法省略
6. }

从实体上我们可以看出，学生和团队是一个多对一得数据关系，这个我们以前都看过，也写过，相信大家都有已经很熟悉了。所以在此具体的配置文件我们也不多写了，我们主要来看一下配置排序的地方，下面我们看一下配置的具体代码：

[html] [view plaincopyprint?](#)

1. <!-- 以名称降序返回 student 集合 -->
2. <set name= table= cascade= order-by=
3. <key column= ></key>
4. <one-to-many class= />
5. </set>

从上面可以看出，其实配置数据库排序很简单，仅仅是在集合标签上配置一个 **order-by** 属性即可。

下面我们就具体来看一下测试代码：

[java] [view plaincopyprint?](#)

1. Transaction t=session.beginTransaction();
- 2.
3. Team team=(Team)session.createQuery("from Team t where t.teamname='team1'").uniqueResult();
4. Set result=team.getStudents();
5. for (Iterator iterator = result.iterator(); iterator.hasNext();) {
6. Student object = (Student) iterator.next();
7. System.out.println(object.getName());

```

8.     }
9.
10.    t.commit();

```

运行这块代码，我们一起来看一下控制台的打印结果：

测试结果：

[sql] [view plaincopyprint?](#)

```

1. <PRE class=html name="code">Hibernate: select team0_.id as id1_, team0_.tea
   mname as teamname1_ from teamOrder team0_ where team0_.teamname='tea
   m1'
2. Hibernate: select students0_.team_id as team4_1_, students0_.id as id1_, student
   s0_.id as id0_0_, students0_.name as name0_0_, students0_.description as desc
   ript3_0_0_, students0_.team_id as team4_0_0_ from studentOrder students0_ w
   here students0_.team_id=? order by students0_.name desc
3.
4.
5. hello
6. default
7. bug</PRE><BR>
8. <BR>
9. <PRE></PRE>
10.<P></P>
11.<PRE></PRE>
12.<P></P>
13.<P><SPAN style="FONT-SIZE: 18px"><SPAN style="COLOR: #ff0000">    </SP
   AN><STRONG><SPAN style="COLOR: #ff0000">从上面输出的 sql 语句就可以看
   出，我们查询到得数据时以 student 表中的 name 进行排序的。所以我们数据库排
   序的配置就这么结束了。</SPAN></STRONG></SPAN></P>
14.<P></P>
15.<P><SPAN style="COLOR: #ff0000; FONT-SIZE: 18px"><STRONG>二. 内存排序
   </STRONG></SPAN></P>
16.<P><SPAN style="FONT-SIZE: 18px"> </SPAN></P>

```

17. `<P>` 内存排序，顾名思义，就是在内存中排序，把查询到得结果加载到内存以后惊醒排序。**Hibernate** 在配置文件中也给我提供了内存排序的配置，那就是 **sort** 属性，它有两个属性值可以直接使用，分别是 **unsorted**（不排序）以及 **natural**（自然排序，即升序），此外，我们还可以自定义排序规则，方式是定义一个类，让其实现 **Comparator** 接口，并且实现该接口中的 **compare** 方法，在该方法中实现排序规则即可。然后将该自定义排序规则的类名作为 **sort** 的属性值即可。 `<set>和`
`<map>元素都具有`
`sort属性，如果设置了该属性，就会对内存中的集合对象进行排序。</P>`
18. `<P><SPAN style="FONT-FAMILY: 宋体`
`"></P>`
19. `<P>` `<set>` 元素的 **sort** 属性为 **natural**，表示对集合中的字符串进行自然排序。**Hibernate** 采用 **org.hibernate.PersistentSortedSet** 作为 **Set** 的实现类，**PersistentSortedSet** 类实现了 **java.util.SortedSet** 接口。当 **Session** 保存一个对象时，会调用 **org.hibernate.type.SortedSetType** 类的 **wrap()** 方法，把对象的集合属性包装为 **SortedSet** 类的实例，下面我们看一下 **wrap()** 方法的源代码如下： `</P>`
20. `<P></P>`
21. `<PRE class=java name="code">public PersistentCollection wrap`
22. `(SessionImplementor session, Object collection) {`
23. `return new PersistentSortedSet`
24. `(session, (java.util.SortedSet) collection);`
25. `} </PRE>
`
26. `
`
27. `<P></P>`
28. `<P>` 从 **wrap()** 方法的源代码看出，应用程序中创建的对象集合属性必须是 **java.util.SortedSet** 类型，否则以上 **wrap()** 方法会抛出 **ClassCastException**。 `` 其实内存排序和数据库排序是一样的，只是配置的参数不同而已，都是在集合标签配置一下，所以在此我们就不以示例演示了。 `</P>`
29. `<P></P>`
30. `<P></P>`

31.<P></P>

32.<P></P>

33.<P>三、组件映射
(component)</P>

34.<P></P>

35.<P> 在 hibernate 中, component 是某个实
体对象的逻辑组成部分, 它与实体的根本区别是, component 是没有标识的, 它是
一个逻辑组成部分, 完全从属于某个实体, 这样就在传统数据库上, 实现了对象的
细粒度划分, 层次分明, 实现了面向对象的领域划分</P>

36.<P>见下图: </P>

37.<P> <IMG alt="" src="http://my.cs
dn.net/uploads/201205/28/1338196835_3738.jpg">

38.</P>

39.<P></P>

40.<P>把
User<SPAN style="FONT-FA
MILY: 宋体">和
Employee<SPAN st
yle="FONT-FAMILY: 宋体">中共同的属性(各种联系方式)拿出来, 放在(抽象到)
到一个单独的类中。这样物理上看有三个类, 不过实体类还是只有
User<SPAN style="
FONT-FAMILY: 宋体">和
Employee<SPAN st
yle="FONT-FAMILY: 宋体">两个, 也就是说数据库里只有
User<SPAN style="
FONT-FAMILY: 宋体">和
Employee<SPAN st
yle="FONT-FAMILY: 宋体">两张表。</P>

41.<P>下面我们就来看一下
user和
contact<SPAN style
="FONT-FAMILY: 宋体">的实体及相关配置: </P>

42.<P>user.java</P>

43.<P></P>

44.<PRE class=java name="code">public class User {

```

45.private int id;
46.private String name;
47.//联系方式
48.private Contact contact;
49.*****set、get 省略
50.}</PRE><BR>
51.<P></P>
52.<P><SPAN style="FONT-SIZE: 18px">contact.java</SPAN></P>
53.<P></P>
54.<P><SPAN style="FONT-SIZE: 18px"></SPAN></P>
55.<PRE class=java name="code">public class Contact {
56.private String address;
57.private String contactTel;
58.private String email;
59.private String zipCode;
60.*****set、get 省略
61.}</PRE><BR>
62.<BR>
63.<P></P>
64.<P><SPAN style="FONT-SIZE: 18px">user.hbm.xml</SPAN></P>
65.<P><SPAN style="FONT-SIZE: 18px"></SPAN></P>
66.<PRE class=html name="code"><?xml version="1.0"?>
67.<!DOCTYPE hibernate-mapping PUBLIC
68."-//Hibernate/Hibernate Mapping DTD 3.0//EN"
69."http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
70.<hibernate-mapping>
71.<class name="com.bjsxt.hibernate.User" table="t_user">
72.<id name="id">
73.<generator class="native"/>
74.</id>
75.<property name="name"/>
76.<component name="contact">
77.<property name="address"/>
78.<property name="contactTel"/>
79.<property name="email"/>

```

```

80.<property name="zipCode"/>
81.</component>
82.</class>
83.</hibernate-mapping></PRE><BR>
84.<BR>
85.<P></P>
86.<P><SPAN style="FONT-SIZE: 18px">    从配置上来看，其实这个地方很好理解，user 把 contact 看做是组成的一部分，只是把他抽出一个单独的实体类了而已，这也正好体现了代码的复用性，其实他就是一对一关系的映射。</SPAN></P>
87.<P></P>
88.<P><SPAN style="FONT-SIZE: 18px">    如果想要生成两张表，hibernate 也提供了相关的配置机制，其实只是换了换标签而已，把 component 标签换成 composite-element，仅此而已，这样就能生成两张表了。</SPAN></P>
89.<P></P>
90.<PRE></PRE>

```

（六十二）细谈 Hibernate（十三）session 缓存机制和三种对象状态

Hibernate 向我们提供的主要的操纵数据库的接口,Session 就是其中的一个,它提供了基本的增,删,改,查方法.而且具有一个缓存机制,能够按照某个时间点,按照缓存中的持久化对象属性的变化来更新数据库,着就是 Session 的缓存清理过程.在 Hibernate 中对象分为三个状态,临时,持久化,游离.如果我们希望 JAVA 里的一个对象一直存在,就必须有一个变量一直引用着这个对象.当这个变量没了.对象也就被 JVM 回收了.这篇博客我们就带大家一起来看一

下 session 的缓存机制，也就是 hibernate 的一级缓存，还有 hibernate 三种对象状态的详细情况。

当 Session 的 `save()` 方法持久化一个 Customer 对象时，Customer 对象被加入到 Session 的缓存中，以后即使应用程序中的引用变量不再引用 Customer 对象，只要 Session 的缓存还没有被清空，Customer 对象仍然处于生命周期中。当 Session 的 `load()` 方法试图从数据库中加载一个 Customer 对象时，Session 先判断缓存中是否已经存在这个 Customer 对象，如果存在，就不需要再到数据库中检索。这样就大大提高了 hibernate 查询的时间效率，只有当事务提交，session 关闭之后，session 缓存才会失效

下面我们来通过一段代码来理解一下 session 缓存：

[java] [view plaincopyprint?](#)

```
1. tx = session.beginTransaction();
2. Customer c1=new Customer("zhangsan",new HashSet());
3. //Customer 对象被持久化，并且加入到 Session 的缓存中
4. session.save(c1);
5. Long id=c1.getId();
6. //c1 变量不再引用 Customer 对象
7. c1=null;
8. //从 Session 缓存中读取 Customer 对象，使 c2 变量引用 Customer 对象
9. Customer c2=(Customer)session.load(Customer.class,id);
10.tx.commit();
11.//关闭 Session，清空缓存
12.session.close();
13.//访问 Customer 对象
14.System.out.println(c2.getName());
15.// c2 变量不再引用 Customer 对象,此时 Customer 对象结束生命周期。
16.c2=null;
```


当 session 调用 save 保存一个对象时，这个对象就被加载到 session 缓存当中，其实调用 save 方法这里有个细节，很多人都忽略了这个细节，就是 save 方法有一个返回值，返回一个 Serializable 接口类型的数据，我们知道像基本数据类型的包装类型都实现了这个接口，其实这个返回值我们可以理解为保存对象的 id，我们在很多时候都能用到这个返回值，这是一个应该注意的地方。当对象被 save 到缓存中时，我们就可以调用对象的 getId 方法来获得他的 id 了。在上面的示例中我们可以看到，虽然 c1 被复位 null 了，但是此时在 session 缓存里面还是有一个变量指向着该对象，所以该对象才不被垃圾回收器回收，当我们在此利用该对象的 id 去用 load 查询时，其实还是去到 session 缓存去找并且返回该对象，当 session 关闭后。缓存清空。

下面我们在来看一个例子，来看一下 get 和 load 的另一个不同点：

[Java] [view plaincopyprint?](#)

```
1. tx = session.beginTransaction();
2. Customer c1=(Customer)session.load(Customer.class,new Long(1));
3. Customer c2=(Customer)session.load(Customer.class,new Long(1));
4. System.out.println(c1==c2); // true or false ??
5. tx.commit();
6. session.close();
```

很明显，这个示例最后打印出来的是 **true**，因为他们获得的是同一个实例，我们具体来分析一下，我们在运行这段代码时，细心的童鞋应该会发现，利用 load 去查询对象时，没有生成 sql 语句，这是为什么呢？既然查询到结果了，为什么没有生成出来 sql 语句呢。这就是我们要说的 load 和 get 方法的第二个不同的地方了，load 方法在查询时，其实是获得的该对象的一

个代理的对象，当我们用到查询到的对象时，他才会去数据库进行查询，如上，如果我们调用 `c1.getName` 方法，这时就会打印出 sql 语句来，这时候他才真正的去数据库查询，而 `get` 方法，他在执行 `get` 的方法的时候就会去数据库查询，产生 sql 语句

Session 缓存的作用

(1) 减少访问数据库的频率。应用程序从内存中读取持久化对象的速度显然比到数据库中查询数据的速度快多了，因此 **Session** 的缓存可以提高数据访问的性能。

(2) 保证缓存中的对象与数据库中的相关记录保持同步。当缓存中持久化对象的状态发生了变化，**Session** 并不会立即执行相关的 **SQL** 语句，这使得 **Session** 能够把几条相关的 **SQL** 语句合并为一条 **SQL** 语句，以便减少访问数据库的次数，从而提高应用程序的性能。

Session 的清理缓存

清理缓存是指按照缓存中对象的状态的变化来同步更新数据库，下面我们还是具体来看一段代码：以下程序代码对 **Customer** 的 **name** 属性修改了两次：

[java] [view plaincopyprint?](#)

```
1. tx = session.beginTransaction();
2. Customer customer=(Customer)session.load(Customer.class,
3. new Long(1));
4. customer.setName("Jack");
5. customer.setName("Mike");
6. tx.commit();
```

当 **Session** 清理缓存时，只需执行一条 **update** 语句：

update CUSTOMERS set NAME= 'Mike'..... where ID=1;

其实第一次调用 setName 是无意义的，完全可以省略掉。

Session 缓存在什么时候才清理呢？我们来看一下：

Session 会在下面的时间点清理缓存：

- 1.当应用程序调用 org.hibernate.Transaction 的 commit()方法的时候，commit()方法先清理缓存，然后再向数据库提交事务。
- 2.当应用程序显式调用 Session 的 flush()方法的时候，其实这个方法我们几乎很少用到，因为我们一般都是在完成一个事务才去清理缓存，提交数据更改，这样我们直接提交事务就可以。

Hibernate 中 Java 对象的三种状态：

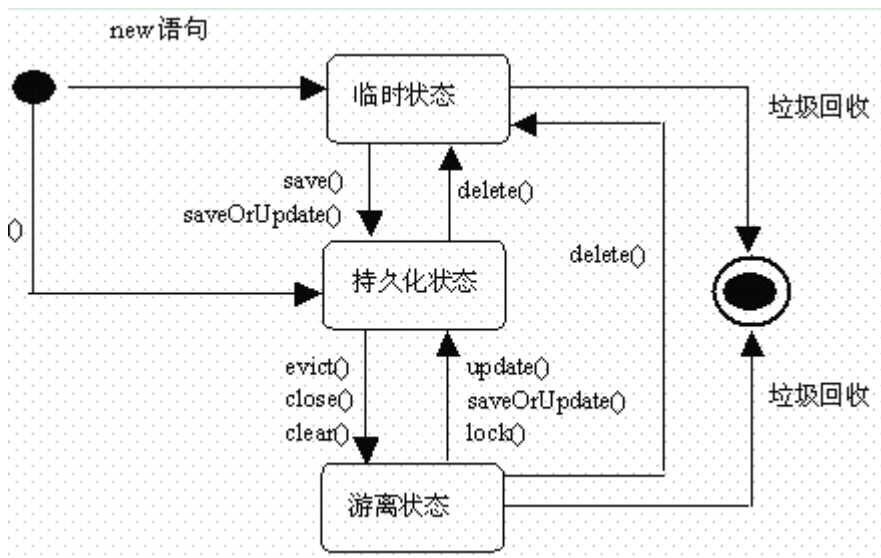
- 1、临时状态（transient）：刚刚用 new 语句创建，还没有被持久化，不处于 Session 的缓存中。处于临时状态的 Java 对象被称为临时对象。
- 2、持久化状态（persistent）：已经被持久化，加入到 Session 的缓存中。处于持久化状态的 Java 对象被称为持久化对象。
- 3、游离状态（detached）：已经被持久化，但不再处于 Session 的缓存中。处于游离状态的 Java 对象被称为游离对象。

持久化状态和临时状态的不同点在于：

- 1、对象持久化状态时，他已经和数据库打交道了，在数据库里面存在着该对象的一条记录。
- 2、持久化状态的对象存在于 session 的缓存当中。
- 3、持久化状态的对象有自己的 OID。

游离状态的对象与持久化状态的对象不同体现在游离状态的对象已经不处于 session 的缓存当中，并且在数据库里面已经不存在该对象的记录，但是他依然有自己的 OID。

对象的状态转换



我们一起来分析一下这个状态转换图，首先一个对象被 **new** 出来之后，他是出于临时状态的，然后调用 **save** 或者 **saveOrUpdate** 方法把对象转换为持久化状态，这里的 **saveOrUpdate** 方法其实是一个偷懒的方法，我们以前用的所有的 **save** 方法的地方都可以修改为该方法，这个方法是在保存数据之前先查看一下这个对象是什么状态，如果是临时状态就保存，如果是游离状态就进行更新。持久化状态转换成游离状态可以是在 session 关闭或者被清理缓存时，在或者就是调用 **evict** 方法，这个方法就是强行把对象从 session 缓存中清除。游离状态转换为持久化状态可以调用 **update** 方法，其实 **update** 方法主要的功能就是把对象从游离状态转换为持久化状态的，因为一般的更新其实不用这个方法也可以。

下面我们举一个具体实例的看一下状态转换过程：

程序代码	Customer 对象的生命周期	Customer 对象的状态
<code>tx = session.beginTransaction(); Customer c1=new Customer("Tom",new HashSet());</code>	开始生命周期	临时状态
<code>session.save(c1);</code>	处于生命周期中	转变为持久化状态
<code>Long id=c1.getId(); c1=null; Customer c2=(Customer)session.load(Customer.class,id); tx.commit();</code>	处于生命周期中	处于持久化状态
<code>session.close();</code>	处于生命周期中	转变为游离状态
<code>System.out.println(c2.getName());</code>	处于生命周期中	处于游离状态
<code>c2=null;</code>	结束生命周期	结束生命周期

这个图需要大家仔细的理解一下，他很好的体现了对象生命周期的进程和对象状态的转换。

下面我们在用一个示例来看一下 **session** 的 **update** 方法是怎么把一个游离状态的对象装换成持久化的：

[java] [view plaincopyprint?](#)

1. `Customer customer=new Customer();`
2. `customer.setName("Tom");`
3. `Session session1=sessionFactory.openSession();`
4. `Transaction tx1 = session1.beginTransaction();`
5. `session1.save(customer);`
6. `tx1.commit();`
7. `session1.close();` //此时 Customer 对象变为游离对象
8. `Session session2=sessionFactory.openSession();`
9. `Transaction tx2 = session2.beginTransaction();`
10. `customer.setName("zhangsan")` //在和 session2 关联之前修改 Customer 对象的属性
11. `session2.update(customer);`
12. `customer.setName("lisi");` //在和 session2 关联之后修改 Customer 对象的属性

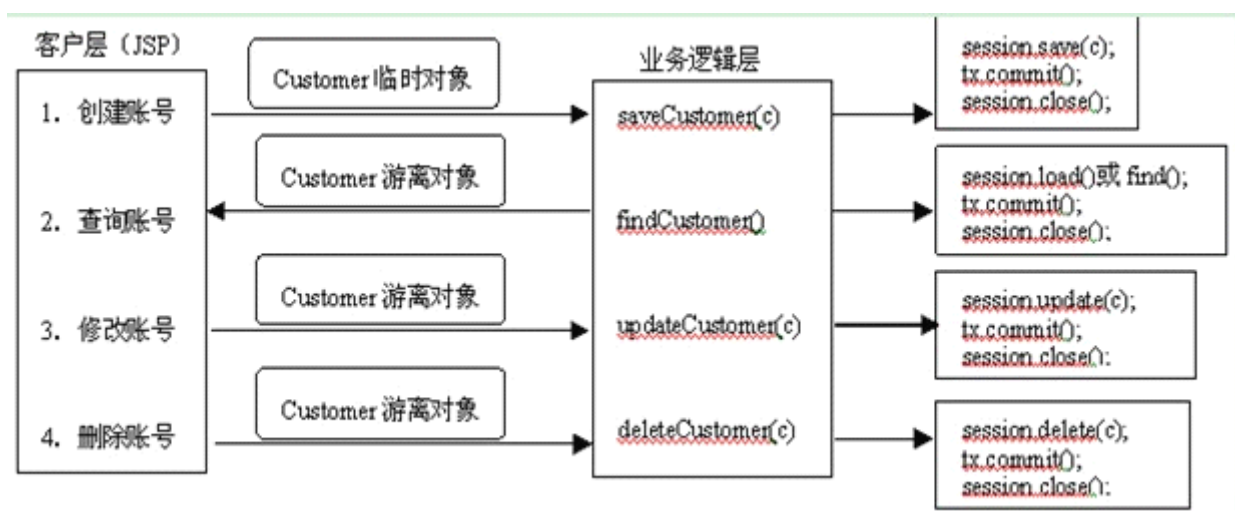
```
13.tx2.commit();  
14.session2.close();
```

当 **session1** 保存完对象，然后事务关闭时，对象就变为游离状态了，此时我们在打开一个 **session**，利用 **update** 方法，在把对象和 **session** 关联起来，然后修改他的属性，提交事务之后，游离状态的对象一样可以修改保存到数据库中，这里虽然修改了两次对象的属性，但只会发送一条 **sql** 语句，因为 **update** 在修改对象数据时，只有在事务提交时，他才会发送 **sql** 语句进行提交。所以只有最后一条修改信息管用。

总结一下 **Session** 的 **update()**方法完成以下操作：

- （1）把 **Customer** 对象重新加入到 **Session** 缓存中，使它变为持久化对象。
- （2）计划执行一个 **update** 语句。值得注意的是，**Session 只有在清理缓存的时候才会执行 update 语句**，并且在执行时才会把 **Customer** 对象当前的属性值组装到 **update** 语句中。因此，即使程序中多次修改了 **Customer** 对象的属性，在清理缓存时只会执行一次 **update** 语句。

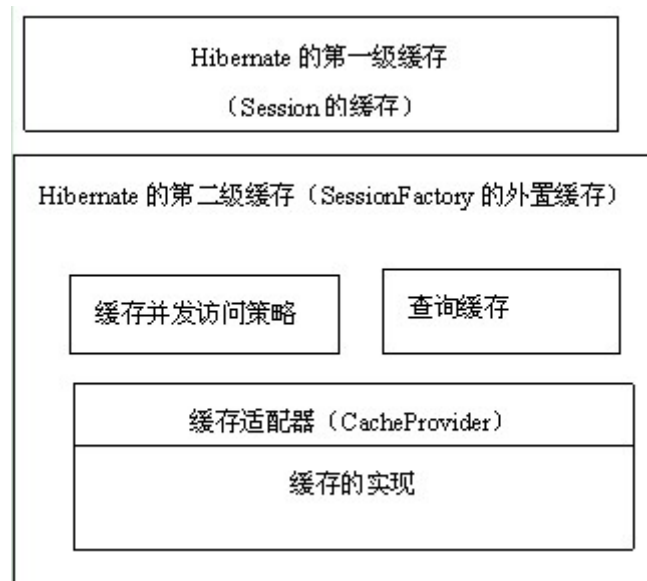
Web 应用程序客户层和业务逻辑层之间传递临时对象和有利对象的过程：



Session 的二级缓存

Hibernate 提供了两级缓存，第一级缓存是 Session 的缓存。由于 Session 对象的生命周期通常对应一个数据库事务或者一个应用事务，因此它的缓存是事务范围的缓存。第一级缓存是必须的，不允许而且事实上也无法被卸除。在第一级缓存中，持久化类的每个实例都具有惟一的 OID。第二级缓存是一个可插拔的缓存插件，它由 SessionFactory 负责管理。由于 SessionFactory 对象的生命周期和应用程序的整个进程对应，因此第二级缓存是进程范围的缓存。这个缓存中存放的是对象的散装数据。第二级缓存是可选的，可以在每个类或每个集合的粒度上配置第二级缓存。

Hibernate 二级缓存结构



关于 **hibernate** 二级缓存的详细内容我们将会在以后的博客中详细介绍，这里我们只是简单介绍一下，详细内容敬请期待以后的文章。。。。

（六十三）细谈 Hibernate （十四）Hibernate 三种检索方式详解

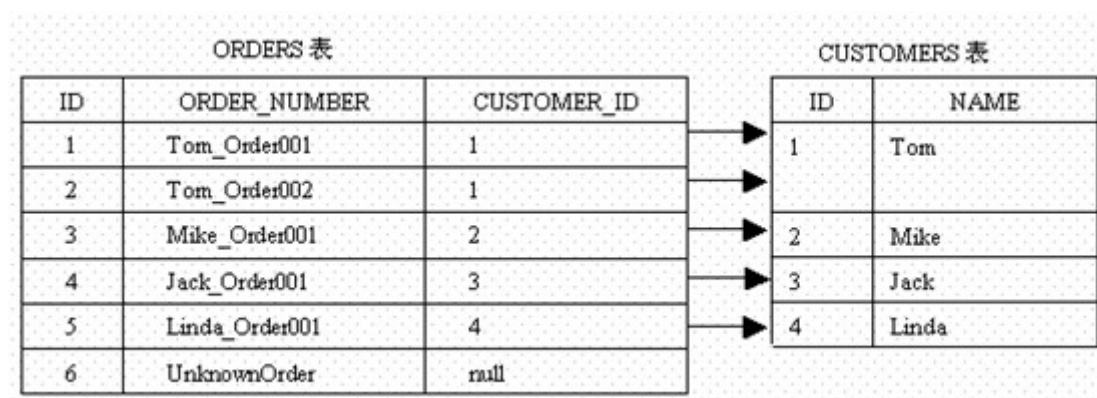
Hibernate 检索机制中主要分为三种，他们各自有各自的好处和缺点，他主要分为以下三种：

1.立即检索策略

2.延迟检索策略

3.左外连接检索策略

立即加载：首先我们来看一下立即加载



[java] [view plaincopyprint?](#)

```
1. List customerLists=session.createQuery("from Customer as c").list();
```

运行以上方法时，Hibernate 将先查询 CUSTOMERS 表中所有的记录，然后根据每条记录的 ID，到 ORDERS 表中查询有参照关系的记录，Hibernate 将依次执行以下 select 语句：

[sql] [view plaincopyprint?](#)

```
1. select * from CUSTOMERS;
2. select * from ORDERS where CUSTOMER_ID=1;
3. select * from ORDERS where CUSTOMER_ID=2;
4. select * from ORDERS where CUSTOMER_ID=3;
```

5. `select * from ORDERS where CUSTOMER_ID=4;`

立即检索缺点:

`select` 语句的数目太多, 需要频繁的访问数据库, 会影响检索性能。

如果需要查询 n 个 `Customer` 对象, 那么必须执行 $n+1$ 次 `select` 查询语句。

这种检索策略没有利用 SQL 的连接查询功能, 例如以上 5 条 `select` 语句完全可以通过以下 1 条 `select` 语句来完成:

[\[java\] view plaincopyprint?](#)

1. `select * from CUSTOMERS left outer join ORDERS`
2. `on CUSTOMERS.ID=ORDERS.CUSTOMER_ID`

以上 `select` 语句使用了 SQL 的左外连接查询功能, 能够在一条 `select` 语句中查询出 `CUSTOMERS` 表的所有记录, 以及匹配的 `ORDERS` 表的记录。在应用逻辑只需要访问 `Customer` 对象, 而不需要访问 `Order` 对象的情况, 加载 `Order` 对象完全是多余的操作, 这些多余的 `Order` 对象白白浪费了许多内存空间。

延迟加载: 我们在一起来看一下延迟加载

一对多, 对于 `<set>` 元素, 应该优先考虑使用延迟检索策略:

[\[html\] view plaincopyprint?](#)

1. `<set name= inverse= lazy= >`

此时运行:

[\[html\] view plaincopyprint?](#)

1. `Customer customer=(Customer)session.get`

2. (Customer.class,new Long(1));

仅立即检索 Customer 对象，执行以下 select 语句：

select * from CUSTOMERS where ID=1;

Customer 对象的 orders 变量引用集合代理类实例，当应用程序第一次访问它，例如调用 customer.getOrders().iterator()方法时，Hibernate 会初始化这个集合代理类实例，在初始化过程中到数据库中检索所有与 Customer 关联的 Order 对象，执行以下 select 语句：

select * from ORDERS where CUSTOMER_ID=1;

访问没有被初始化的游离状态的集合代理类实例

[java] [view plaincopyprint?](#)

1. Session session=sessionFactory.openSession();
2. tx = session.beginTransaction();
3. Customer customer=(Customer)session.get(Customer.class,new Long(1));
4. tx.commit();
5. session.close();
6. //抛出异常
7. Iterator orderIterator=customer.getOrders().iterator();
8. 执行以上代码，会抛出以下异常：
9. ERROR LazyInitializer:63 - Exception initializing proxy

优点

由应用程序决定需要加载哪些对象，可以避免执行多余的 select 语句，以及避免加载应用程序不需要访问的对象。因此能提高检索性能，并且能节省内存空间。

缺点

应用程序如果希望访问游离状态的代理类实例，必须保证它在持久化状态时已经被初始化。

适用范围

一对多或者多对多关联。

应用程序不需要立即访问或者根本不会访问的对象。

左外连接检索策略

默认情况下，多对一关联使用左外连接检索策略。

如果把 Order.hbm.xml 文件的<many-to-one>元素的 outer-join 属性设为 true，总是使用左外连接检索策略。

•对于以下程序代码：

[java] [view plaincopyprint?](#)

```
1. Order order=(Order)session.get(Order.class,new Long(1));
```

在运行 session.get()方法时，Hibernate 执行以下 select 语句：

[sql] [view plaincopyprint?](#)

```
1. select * from ORDERS left outer join CUSTOMERS
2. on ORDERS.CUSTOMER_ID=CUSTOMERS.ID where ORDERS.ID=1
```

左外连接查询优点

1. 对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便的从一个对象导航到与它关联的对象。
2. 使用了外连接，select 语句数目少。

左外连接查询缺点

1. 可能会加载应用程序不需要访问的对象，白白浪费许多内存空间。

2. 复杂的数据库表连接也会影响检索性能。

左外连接查询适用范围

1. 多对一或者一对一关联。
2. 应用程序需要立即访问的对象。
3. 数据库系统具有良好的表连接性能

在映射文件中设定的检索策略是固定的，要么为延迟检索，要么为立即检索，要么为外连接检索。但应用逻辑是多种多样的，有些情况下需要延迟检索，而有些情况下需要外连接检索。Hibernate 允许在应用程序中覆盖映射文件中设定的检索策略，由应用程序在运行时决定检索对象图的深度。

以下 Session 的方法都用于检索 OID 为 1 的 Customer 对象：

[html] [view plaincopyprint?](#)

1. session.createQuery("from Customer as c where c.id= ");
2. session.createQuery("from Customer as c left join fetch c.orders
3. where c.id= ");

- 在执行第一个方法时，将使用映射文件配置的检索策略。
- 在执行第二个方法时，在 HQL 语句中显式指定左外连接检索关联的 Order 对象，因此会覆盖映射文件配置的检索策略。不管在 Customer.hbm.xml 文件中<set>元素的 lazy 属性是 true 还是 false, Hibernate 都会执行以下 select 语句：

[sql] [view plaincopyprint?](#)

1. select * from CUSTOMERS left outer join ORDERS
2. on CUSTOMERS.ID =ORDERS.CUSTOMER_ID
3. where CUSTOMERS.ID=1;

（六十四）细谈 Hibernate（十五）HQL 与 QBC 查询方式详解

首先来看一下，hibernate 提供的几种检索方式：

1.导航对象图检索方式：根据已经加载的对象，导航到其他对象。例如，对于已经加载的 **Customer** 对象，调用它的 `getOrders().iterator()`方法就可以导航到所有关联的 **Order** 对象，假如在关联级别使用了延迟加载检索策略，那么首次执行此方法时，**Hibernate** 会从数据库中加载关联的 **Order** 对象，否则就从缓存中取得 **Order** 对象。

2、OID 检索方式：按照对象的 **OID** 来检索对象。**Session** 的 `get()`和 `load()`方法提供了这种功能。如果在应用程序中事先知道了 **OID**，就可以使用这种检索对象的方式。

3、HQL 检索方式：**Hibernate** 提供了 **Query** 接口，它是 **Hibernate** 提供的专门的 **HQL** 查询接口，能够执行各种复杂的 **HQL** 查询语句。

4、QBC 检索方式：使用 **QBC**（**Query By Criteria**）**API** 来检索对象。这种 **API** 封装了基于字符串形式的查询语句，提供了更加面向对象的接口。

前两种在前面我们都熟练的用过多次了，现在我們来看一下 **HQL** 检索方式。**HQL**（**Hibernate Query Language**）是面向对象的查询语言，它和 **SQL** 查询语言有些相似。在 **Hibernate** 提供的各种检索方式中，**HQL** 是使用最广的一种检索方式。它具有以下功能：

—在查询语句中设定各种查询条件

- 支持投影查询，即仅检索出对象的部分属性
- 支持分页查询
- 支持连接查询
- 支持分组查询，允许使用 **having** 和 **group by** 关键字
- 提供内置聚集函数，如 **sum()**、**min()**和 **max()**
- 支持子查询，即嵌入式查询
- 支持动态绑定参数

下面我们通过一个实例来看一下 HQL 检索的步骤：

[java] [view plaincopyprint?](#)

```
1. //创建一个 Query 对象
2. Query query=session.createQuery("from Customer as c where "
3. +" c.name=:customerName "
4. +"and c.age=:customerAge");
5. //动态绑定参数
6. query.setString("customerName","Tom");
7. query.setInteger("customerAge",21);
8. //执行查询语句，返回查询结果
9. List result= query.list();
```

从上面这个实例我们可以看出：

（1）通过 **Session** 的 **createQuery()**方法创建一个 **Query** 对象，它包含一个 HQL 查询语句。HQL 查询语句可以包含命名参数，如“**customerName**”和“**customerAge**”都是命名参数。

(2) 动态绑定参数。Query 接口提供了给各种类型的命名参数赋值的方法，例如 `setString()` 方法用于为字符串类型的 `customerName` 命名参数赋值。

(3) 调用 Query 的 `list()` 方法执行查询语句。该方法返回 List 类型的查询结果，在 List 集合中存放了符合查询条件的持久化对象。

Query 中还提供了良好的方法连编程方式，查看 hibernate 的 API 我们可以发现，query 类提供的 `set` 方法的返回值还是 query 对象，这样可以使代码更为简介：例如。

[java] [view plaincopyprint?](#)

```
1. List result=session.createQuery(".....") .setString("customerName","Tom")
2. .setInteger("customerAge",21) .list();
```

QBC

采用 HQL 检索方式时，在应用程序中需要定义基于字符串形式的 HQL 查询语句。QBC API 提供了检索对象的另一种方式，它主要由 Criteria 接口、Criterion 接口和 Expression 类组成，它支持在运行时动态生成查询语句。

QBC 检索步骤：

- 1) 调用 Session 的 `createCriteria()` 方法创建一个 Criteria 对象。
- 2) 设定查询条件。Expression 类提供了一系列用于设定查询条件的静态方法，这些静态方法都返回 Criterion 实例，每个 Criterion 实例代表一个查询条件。Criteria 的 `add()` 方法用于加入查询条件。

3)调用 **Criteria** 的 **list()**方法执行查询语句。该方法返回 **List** 类型的查询结果, 在 **List** 集合中存放了符合查询条件的持久化对象。下面还是通过一个实例来看一下 **QBC** 查询的具体步骤:

[java] [view plaincopyprint?](#)

```
1. //创建一个 Criteria 对象
2. Criteria criteria=session.createCriteria(Customer.class);
3. //设定查询条件, 然后把查询条件加入到 Criteria 中
4. Criterion criterion1= Expression.like("name", "T%");
5. Criterion criterion2= Expression.eq("age", new Integer(21));
6. criteria=criteria.add(criterion1);
7. criteria=criteria.add(criterion2);
8. //执行查询语句, 返回查询结果
9. List result=criteria.list();
```

对于以上程序代码, 当运行 **Criteria** 的 **list()**方法时, **Hibernate** 执行的 **SQL** 查询语句为:

```
select * from CUSTOMERS where NAME like 'T%' and AGE=21;
```

注: **Expression**; 类是 **org.hibernate.criterion.Restrictions** 的子类, 所以也可以换成该类。**Criteria** 也是采用的方法连的编程风格, 因为 **add** 方法返回值也是 **Criteria**, 就像下面这段代码:

[java] [view plaincopyprint?](#)

```
1. List result=session.createCriteria(Customer.class).add(Expression.like("name", "T%"))
   .add(Expression.eq("age", new Integer(21))).list();
```

Query 和 **Criteria** 接口都提供了用于分页显示查询结果的方法:

–**setFirstResult(int firstResult)**: 设定从哪一个对象开始检索，参数 **firstResult** 表示这个对象在查询结果中的索引位置，索引位置的起始值为 0。默认情况下，**Query** 和 **Criteria** 接口从查询结果中的第一个对象，也就是索引位置为 0 的对象开始检索。

–**setMaxResult(int maxResults)**: 设定一次最多检索出的对象数目。默认情况下，**Query** 和 **Criteria** 接口检索出查询结果中所有的对象。

分页查询：

采用 **HQL** 检索方式：

[java] [view plaincopyprint?](#)

```
1. Query query = session.createQuery("from  
2. Customer c  
3. order by c.name asc");  
4. query.setFirstResult(0);  
5. query.setMaxResults(10);  
6. List result = query.list();
```

采用 **QBC** 检索方式

[java] [view plaincopyprint?](#)

```
1. Criteria criteria = session.createCriteria(  
2. Customer.class);  
3. criteria.addOrder( //criteria 里面提供添加排序方式的方法  
4. Order.asc("name") );  
5. criteria.setFirstResult(0);  
6. criteria.setMaxResults(10);  
7. List result = criteria.list ( )
```

上面我们大体了解了 hql 语句和 QBC 两种查询方式,具体的用法我们来看一下, 首先看一下 HQL.

我们平时查询的时候, HQL 语句总是” from 类名”, 这样查询, 这是我们要查询出对应对象的所有属性。当我们只是想查询实体类的其中的几个字段, 而不是整个实体类的对象时, 我们就需要把前面省略的 hql 语句加上了。

例如 HQL 语句: `select s.name,s.age from student s`, 他代表只是查询 `student` 表中的 `name` 和 `age` 属性, 其他的不要。这时调用 `query.list()`

`()` 方法时返回的不是一个个对象了, 因为我们查询的只是对象的一部分属性。那查询到的 `list` 集合里是什么东东呢。其实 `list` 对象中里面每一个元素是一个 `object` 对象的数组, 每一个数组包含查询对象的几个字段。这些数据只是一些游离的数据。我们想获得的字段数据可以用下面方式去遍历。

代码如下:

```
tx = session.beginTransaction();

Query query = session.createQuery("select s.name, s.age from Student s'

List list = query.list();

for(int i = 0; i < list.size(); i++)
{
    Object[] obj = (Object[])list.get(i);
    System.out.println(obj[0] + ", " + obj[1]);
}
```

以上情况还可以用另一种方式来解决但这种方式成功执行的前提是, 在 `student` 实体类里面构造了包含 `name` 和 `age` 两个属性的构造方法。

[java] [view plaincopyprint?](#)

1. `Query query = session.createQuery("select new Student(s.name, s.age) from Student s");`
2. `List list = query.list();`

```

3. for(int i = 0; i < list.size(); i++)
4. {Student student = (Student)list.get(i)    System.out.println(student.getName() + ",
      " +          student.getAge());
5. System.out.println(student.getCardId());//此处返回空，数据库没有查该字段
6. }

```

通过上面 HQL 语句我们可以看出，这时查询出来的就是一个一个 **student** 对象了，而不是一个个游离的数据了，这里一定要注意 HQL 语句的写法。这种写法在 **sql** 语句中肯定是不支持的。

HQL 语句进行内连接查询：**from Team t join t.student;**这条内连接语句意思为让 **team** 表和 **student** 表进行内连接查询。很多同学回想没有 **on** 语句，查询条件是什么呢？这是由于 **hbm** 文件已经注明了表之间的字段连接关系，所以他会自动去 **on** 连接。下面看一下自动生成的 **sql** 语句大家就没白了。

Hibernate 自动生成的查询语句为：

```

Hibernate:
  select
    team0_.id as id3_0_,
    students1_.id as id0_1_,
    team0_.teamName as teamName3_0_,
    students1_.name as name0_1_,
    students1_.cardId as cardId0_1_,
    students1_.age as age0_1_
  from
    team team0_
  inner join
    student students1_
    on team0_.id=students1_.team_id I

```

Hibernate 配置给我们提供了一个格式化 **sql** 语句的配置，配置 **sql_format** 的属性为 **true**，就会向上面那样，得到格式化的 **sql** 语句。

在内连接查询结果中，刚开始我有这么一个疑问，连接查询得到的结果集是两个表中的并集集合，那得到的对象到底是哪一个呢？不是 **student**。是不是 **team**。原来，**hibernate** 早就给我们准备了这个东西。当我们调用 **list** 方法获得结果的集合时，得到的集合实际上是 **object** 数组的集合。这个数组里面一般有两个对象，像上面那条 **hql** 语句就是得到的 **student** 对象和 **team** 对象数组的集合，下面我们就看一下具体的代码实现：

[java] [view plaincopyprint?](#)

```
1. Query query = session.createQuery("from Team t join t.students");
2. List list = query.list();
3. for(int i = 0; i < list.size(); i++)
4. {
5.     Object[] obj = (Object[]) list.get(i);
6.     Team team = (Team)obj[0];
7.     Student student = (Student)obj[1];
8.     System.out.println(team.getTeamName());
9.     System.out.println(student.getName());
10.}
```

内连接中还深藏着一个不为人知的秘密，当我们把延迟加载设为 **true**，也就是让他延迟加载对应对象信息时，我们在 **session** 关闭之后去遍历 **list** 集合，这时你会发现一样的加载出来对应的 **student** 的信息，**这时因为内连接查询把配置文件中的延迟加载信息给覆盖掉了。在 session 关闭之前就把对应的对象信息给加载出来了。**

hql 中实体参数设置

下面我们在来看一下 **hibernate** 用 **hql** 中实体参数设置，所谓实体参数设置就是在利用 **HQL** 语句执行操作时，对 **HQL** 语句中设置参数，以前我们接触

到的都是一些基本数据类型的参数设置，hibernate 中都提供了相应的 set 方法，下面我们来看一下下面一个 hql 查询：

[java] [view plaincopyprint?](#)

```
1. Team team=session.get(Team.class,1);
2. Query query = session.createQuery("from student s where s.team=:team and s.age>20");
```

在这个地方，我们查询的条件为学生对应的 team 对象与刚查询出来的对象相等。在 hibernateAPI 中提供了两种方式设置这个地方的参数，下面我们一一来看一下：

1.第一种：

query.setParameter("team",team,Hibernate.entity(Team.class));这种方式需要三个参数，第一个参数是指定设定的参数，这里设置的是上面 hql 语句中的 team 参数，第二个参数是设置参数的值，第三个参数是一个 Type 类型的数据，hibernate 中提供了一个 Hibernate 类，他里面包含了大量的静态工具方法，其中就有获得 Type 对象的方法 entity。

2.第二种：

query.setEntity("team",team),这个方法比较简单，并且比较实用，我们在大多数的时候都使用这一个，他和设置基本数据类型的 set 方法比较相识。直接设置参数名和参数值就 OK 了。

值得注意的是，这里的 hql 语句这样比较两个对象是否相等，其实在 hibernate 底层会转换成对应的外键关联。

过滤：

hibernateAPI 提供了一个起到过滤器作用的方法,这个方法就是 createFilter (object collection, string s) 方法,他有两个参数,第一个参数是指所要过滤的对象,也就是集合对象,第二个参数指的是过滤语句,其实就是 hql 语句的一部分:例如下面这段代码:

[java] [view plaincopyprint?](#)

1. Query query = session.createFilter(team.getStudents(), "where age > 20");
2. List<Student> list = query.list();

QBC: 作为真正的面向对象对数据库进行操作的一种方式,其内部提供了大量的 **API** 对数据库操作和条件,下面我们根据具体示例来看一下其 **API** 的用法:

[java] [view plaincopyprint?](#)

1. //增加年龄限制 12-30 岁
2. Criteria criteria = session.createCriteria(Student.class).add(
3. Restrictions.between("age", new Integer(12), new Integer(30)));
4. //增加 name 条件限制, 名字以 t 开头的
5. Criteria criteria = session.createCriteria(Student.class).add(
6. Restrictions.like("name", "t%"));
7. //增加范围限制, 名字必须是"jerry", "spark", "tom"中的一个
8. String[] names = { "jerry", "spark", "tom" };
9. Criteria criteria = session.createCriteria(Student.class).add(
10. Restrictions.in("name", names));
- 11.//对查询到的数据进行排序, 以 age 升序, 以 cardid 降序
- 12.Criteria criteria = session.createCriteria(Student.class).addOrder(
- 13.Order.asc("age")).addOrder(Order.desc("cardId"));
- 14.//执行数据查询
- 15.List<Student> list = criteria.list();

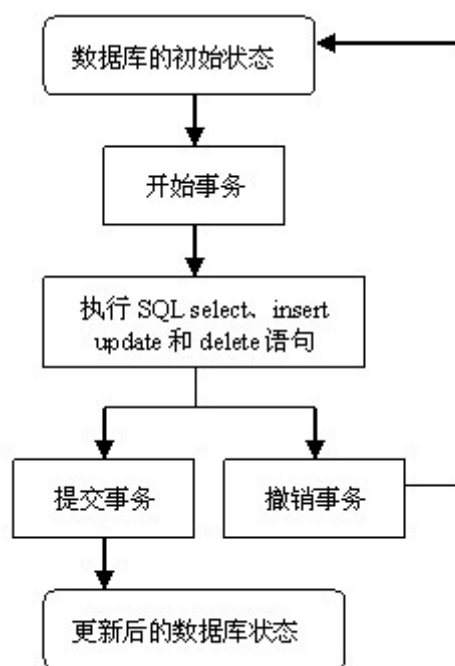
（六十五）细谈 Hibernate （十六）数据库事务与隔离级别

数据库事务：事务是指一组相互依赖的操作行为，如银行交易、股票交易或网上购物。事务的成功取决于这些相互依赖的操作行为是否都能执行成功，只要有一个操作行为失败，就意味着整个事务失败。关于事务的一个经典例子就是：**A** 到银行办理转账事务，把 100 元钱转到 **B** 的账号上，这个事务包含以下操作行为：

- （1）从 **A** 的账户上减去 100 元。
- （2）往 **B** 的账户上增加 100 元。

显然，以上两个操作必须作为一个不可分割的工作单元。假如仅仅第一步操作执行成功，使得 **Tom** 的账户上扣除了 100 元，但是第二步操作执行失败，**Jack** 的账户上没有增加 100 元，那么整个事务失败。数据库事务是对现实生活中事务的模拟，它由一组在业务逻辑上相互依赖的 **SQL** 语句组成。

下面我们一起来看一下**数据库事务的生命周期：**



这个数据库事务的生命周期图反应出数据库事务的三个边界：

- 1.事务的开始边界。
- 2.事务的正常结束边界（COMMIT）：提交事务，永久保存被事务更新后的数据库状态。
- 3.事务的异常结束边界（ROLLBACK）：撤销事务，使数据库退回到执行事务前的初始状态。

其实每个数据库连接都有个全局变量@@autocommit，表示当前的事务模式，它有两个可选值：0：表示手工提交模式。1：默认值，表示自动提交模式。

在自动提交模式下，每个 SQL 语句都是一个独立的事务。也就是说，每执行一条 sql 语句，数据库都会自动提交这个事务，当我们用数据库另一个客户端去查询的时候，我们可以看到这个新修改或插入的数据。在手工提交模式下，必须显式指定事务开始边界和结束边界：

–事务的开始边界: **begin**

–提交事务: **commit**

–撤销事务: **rollback**

下面我们来看一下**通过 JDBC API 是如何声明事务边界的:**

Connection 提供了以下用于控制事务的方法:

1.**setAutoCommit(boolean autoCommit):** 设置是否自动提交事务

2.**commit():** 提交事务

3.**rollback():** 撤销事务

下面我们看一下具体的应用示例:

[java] [view plaincopyprint?](#)

```
1. try {
2.   con = java.sql.DriverManager.getConnection(dbUrl,dbUser,dbPwd);
3.   //设置手工提交事务模式
4.   con.setAutoCommit(false);
5.   stmt = con.createStatement();
6.   //数据库更新操作 1
7.   stmt.executeUpdate("update ACCOUNTS set BALANCE=900 where ID=1 ");
8.   //数据库更新操作 2
9.   stmt.executeUpdate("update ACCOUNTS set BALANCE=1000 where ID=2 ");
10.  con.commit(); //提交事务
11. }catch(Exception e) {
12. try{
13.  con.rollback(); //操作不成功则撤销事务
14. }catch(Exception ex){
15. //处理异常
16. ....
17. }
18. //处理异常
19. ....
```

20.}finally{...}

看到上边的示例我们可以看出，其实 **hibernate** 事务边界就是模仿者 **JDBC** 的事务边界来的，其实在 **hibernate** 底层的事务管理就是利用的 **JDBC** 的事务管理。我们来看一下 **hibernate** 事务边界：

1.声明事务的开始边界：

`Transaction tx=session.beginTransaction();`

2.提交事务: `tx.commit();`

3.撤销事务: `tx.rollback();`

我们在学习 **JDBC** 数据库事务管理的时候，重点也是难点的学习了 **jdbc 多个事务并发问题**。既然 **hibernate** 底层是用 **JDBC** 事务管理实现的，那么它也一定存在着多个事务并发的问题。下面我们就具体来看一下：

hibernate 多个事务并发的并发问题：

- 第一类丢失更新：撤销一个事务时，把其他事务已提交的更新数据覆盖。
- 脏读：一个事务读到另一事务未提交的更新数据。
- 虚读：一个事务读到另一事务已提交的新插入的数据。
- 不可重复读：一个事务读到另一事务已提交的更新数据。
- 第二类丢失更新：这是不可重复读中的特例，一个事务覆盖另一事务已提交的更新数据。

下面我们就脏读来举一个示例：

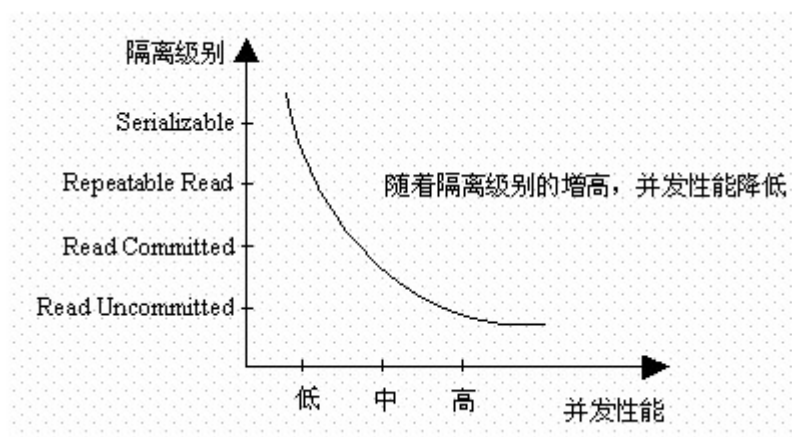
时间	取款事务	支票转账事务
T1	开始事务	
T2		开始事务
T3	查询账户的存款余额为 1000 元。	
T4		
T5	取出 100 元，把存款余额改为 900 元。	
T6		查询账户的存款余额为 900 元。（脏读）
T7	撤销该事务，把存款余额恢复为 1000 元。	
T8		汇入 100 元，把存款余额改为 1000 元。
T9		提交事务

取款事务在 T5 时刻把存款余额改为 900 元，支票转账事务在 T6 时刻查询账户的存款余额为 900 元，取款事务在 T7 时刻被撤销，支票转账事务在 T8 时刻把存款余额改为 1000 元。由于支票转账事务查询到了取款事务未提交的更新数据，并且在这个查询结果的基础上进行更新操作，如果取款事务最后被撤销，会导致银行客户损失 100 元。

事务隔离级别

关于事务隔离级别，我们来看一下下面的两个图解：

隔离级别	是否出现第一类 丢失更新	是否出现脏读	是否出现虚读	是否出现不可 重复读	是否出现第二类 丢失更新
Serializable	否	否	否	否	否
Repeatable Read	否	否	是	否	否
Read Committed	否	否	是	是	是
Read Uncommitted	否	是	是	是	是



由上图可以看出：隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为 **Read Committed**，它能够避免脏读，而且具有较好的并发性能。尽管它会导致不可重复读、虚读和第二类丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

下面我们就具体来看一下 **hibernate** 怎么来配置隔离级别：在 **Hibernate** 的配置文件中可以显式的设置隔离级别。每一种隔离级别都对应一个整数：

- 1: Read Uncommitted
- 2: Read Committed
- 4: Repeatable Read
- 8: Serializable

例如，以下代码把 **hibernate.cfg.xml** 文件中的隔离级别设为

Read Committed：

hibernate.connection.isolation=2

对于从数据库连接池中获得的每个连接，**Hibernate** 都会把它改为使用

Read Committed 隔离级别。

分享到：

（六十六）细谈 struts2（十二）struts2 国际化底层大揭秘

Struts2 的博客在前不久已经停止了，但是里面还有很多内容我们都还没接触到，所以现在我们在补充一下 struts2 的内容。这篇博客我们主要是一块来看一下 struts2 内对国际化的支持。在了解 struts2 对资源国际化支持之前，我们先来看一下 JDK 对国际化的支持，因为如果你看一下啊源码你可以发现，其实 struts2 中国际化的支持底层主要就是对 JDK 中提供的国际化的一个封装。

一：JDK 对国际化的支持

所谓国际化，就是我们写的应用程序在不同的地域和支持不同语言的地方可以给用户一个用户所在地域的语言支持。也就是说我在中国你的应用程序就给我中国的提示，我在美国你就给我英语的提示。在看具体应用之前我们先来熟悉一下几个 JDK 中提供资源国际化的类：

1.Locale 类

Locale 对象表示了特定的地理、政治和文化地区。需要 Locale 来执行其任务的操作称为语言环境敏感的操作，它使用 Locale 为用户量身定制信息。例如，显示一个数值就是语言环境敏感的操作，应该根据用户的国家、地区或文化的风俗/传统来格式化该数值。

使用此类中的构造方法来创建 Locale：

[java] [view plaincopyprint?](#)

1. Locale(String language)根据语言创建对象

[java] [view plaincopyprint?](#)

1. Locale(String language, String country)根据语言和国家来创建对象

1. `Locale(String language, String country, String variant)`根据语言、国家/地区和变量构造一个语言环境。

语言参数是一个有效的 ISO 语言代码。这些代码是由 ISO-639 定义的小写两字母代码。在许多网站上都可以找到这些代码的完整列表，如：

<http://www.loc.gov/standards/iso639-2/englangn.html>。 `Locale` 类提供了一些方便的国家常量，可用这些常量为常用的语言环境创建 `Locale` 对象。例如，下面的内容为美国创建了一个 `Locale` 对象：`Locale.US`。

创建完 `Locale` 后，就可以查询有关其自身的信息。使用 `getCountry` 可获取 ISO 国家/地区代码，使用 `getLanguage` 则获取 ISO 语言代码。可用使用 `getDisplayCountry` 来获取适合向用户显示的国家/地区名。同样，可用使用 `getDisplayLanguage` 来获取适合向用户显示的语言名。有趣的是，`getDisplayXXX` 方法本身是语言环境敏感的，它有两个版本：一个使用默认的语言环境作为参数，另一个则使用指定的语言环境作为参数。

2.ResourceBundle

资源包包含特定于语言环境的对象。当程序需要一个特定于语言环境的资源时（如 `String`），程序可以从适合当前用户语言环境的资源包中加载它。简单来说就是通过静态方法来获得程序外界的资源包。使用这种方式，可以编写很大程度上独立于用户语言环境的程序代码，它将资源包中大部分（即便不是全部）特定于语言环境的信息隔离开来。

这使编写的程序可以：

- 轻松地本地化或翻译成不同的语言
- 一次处理多个语言环境
- 以后可以轻松进行修改，以便支持更多的语言环境

当程序需要特定于语言环境的对象时，它使用 `getBundle` 方法加载 `ResourceBundle` 类：

[java] [view plaincopyprint?](#)

```
1. ResourceBundle myResources =  
2.     ResourceBundle.getBundle("MyResources", currentLocale);
```

这里我们有必要说一下这里的两个参数，第一个参数指定我们外部资源文件的文件名的头，为什么说是文件头呢，要想知道这个我们还得先说一些我们的资源文件的命名规则是：文件头_语言代号_国家代号.properties。这里的文件名是我们自己起的，语言代号和国家代号都是一些定义好的，我们直接去调用就 OK 了。当程序所要找的语言环境我们没有定义的话，他会默认的去找:文件头.properties,第二个参数是设置我们的地域信息。

资源包包含键/值对。键唯一地标识了包中特定于语言环境的对象。看到这里相信大家都应该想到这里的资源用什么文件最合适了，对，就是 **properties** 文件比较合适，JDK 中提供了很多重载的 `getBundle` 方法，具体的看下图：

static ResourceBundle	getBundle (String baseName) 使用指定的基本名称、默认的语言环境和调用者的类加载器获取资源包。
static ResourceBundle	getBundle (String baseName, Locale locale) 使用指定的基本名称、语言环境和调用者的类加载器获取资源包。
static ResourceBundle	getBundle (String baseName, Locale locale, ClassLoader loader) 使用指定的基本名称、语言环境和类加载器获取资源包。
static ResourceBundle	getBundle (String baseName, Locale targetLocale, ClassLoader loader, ResourceBundle.Control control) 使用指定基本名称、目标语言环境、类加载器和控件返回资源包。
static ResourceBundle	getBundle (String baseName, Locale targetLocale, ResourceBundle.Control control) 使用指定基本名称、目标语言环境和控件、调用者的类加载器返回一个资源包。
static ResourceBundle	getBundle (String baseName, ResourceBundle.Control control) 使用指定基本名称、默认语言环境和指定控件返回一个资源包。
abstract Enumeration < String >	getKeys () 返回键的枚举。

得到 **ResourceBundle** 对象之后，我们可以调用他的一些内置的 **getxx** 方法获得其相应的信息

abstract Enumeration < String >	getKeys () 返回键的枚举。
Locale	getLocale () 返回此资源包的语言环境。
Object	getObject (String key) 从此资源包或它的某个父包中获取给定键的对象。
String	getString (String key) 从此资源包或它的某个父包中获取给定键的字符串。
String []	getStringArray (String key) 从此资源包或它的某个父包中获取给定键的字符串数组。

3.MessageFormat

在国际化中的资源配置文件中我们也经常会看到占位符的形式出现，JDK 中之所以可以支持占位符，完全就是看 **MessageFormat** 这个类的支持。当然了，这个也只是这个类的其中一个功能。**MessageFormat** 提供了以与语言无关方式生成连接消息的方式。使用此方法构造向终端用户显示的消息。**MessageFormat** 获取一组对象，格式化这些对象，然后将格式化后的字符串插入到模式中的适当位置。

注：**MessageFormat** 不同于其他 **Format** 类，因为 **MessageFormat** 对象是用其构造方法之一创建的（而不是使用 **getInstance** 样式的工厂方法创建的）。工厂方法不是必需的，因为 **MessageFormat** 本身不实现特定于语言环境的行为。特定于语言环境的行为是由所提供的模式和用于已插入参数的子格式来定义的。

MessageFormat 类提供了两个构造方法，下面我们来看一下：

MessageFormat(String pattern)

构造默认语言环境和指定模式的 **MessageFormat**。

MessageFormat(String pattern, Locale locale)

构造指定语言环境和模式的 **MessageFormat**。

在获取到资源信息之后在进行格式化设置的时候我们一般不去 **new** 出他的对象，而是用他的一个静态方法：**format** 方法，我们来看一下他提供的一些重载方法：

format(Object[] arguments, StringBuffer result, FieldPosition pos)

格式化一个对象数组，并将 **MessageFormat** 的模式添加到所提供的 **StringBuffer**，用格式化后的对象替换格式元素。

format(Object arguments, StringBuffer result, FieldPosition pos)

格式化一个对象数组，并将 **MessageFormat** 的模式添加到所提供的 **StringBuffer**，用格式化后的对象替换格式元素

format(String pattern, Object... arguments)

创建具有给定模式的 **MessageFormat**，即为带有占位符的值。并用它来格式化给定的参数。

好了，几个重要的类介绍完了以后，我相信大家对资源国际化一定有和一个深刻的了解了吧。下面我们就以一个简单的小实例来把这几个类的用法串起来：

[java] [view plaincopyprint?](#)

```
1. Locale locale = Locale.US;
2. ResourceBundle bundle = ResourceBundle
3. .getBundle("caoshenghuan", locale);
4. String value = bundle.getString("hello");
5. String result = MessageFormat.format(value, new Object[] { "曹胜欢" });
6. System.out.println(result);
```

我们可以看到，代码很简单，就是获取外部资源文件的 **key** 为 `hello` 的资源值。下面我们看一下资源文件的写法：

[plain] [view plaincopyprint?](#)

```
1. caoshenghuan_en_US.properties:
2. hello=helloworld{0}
```

Struts2 国际化是建立在 **Java** 国际化的基础上的，一样是通过提供不同国家/语言环境的消息资源，然后通过 **ResourceBundle** 加载指定 **Locale** 对应的资源文件，再取得该资源文件中指定 **key** 对应的消息--整个过程与 **JAVA** 程序的国家化完全相同，只是 **Struts2** 框架对 **JAVA** 程序国际化进行了进一步封装，从而简化了应用程序的国际化。

Struts2 需要国际化的部分

1.类型转换:

2.数据校验:

3.验证框架 xml 配置文件的国际化: RegisterAction-validation.xml 文件

```
<message key="username.xml.invalid"/>
```

4.JSP 页面的国际化: <s:text name="addUser"/>

5.Action 的国际化:利用 ActionSupport 类提供的 getText()方法.

6.Struts2 中加载全局资源文件

struts.xml

```
<constant name="struts.custom.i18n.resources" value="baseName"/>
```

或

struts.properties

struts.custom.i18n.resources=baseName

访问国际化消息

Struts2 访问国际化消息主要有如下三种方式:

(1)JSP 页面: <s:text name="key"/>

(2)Action 类中: 使用 ActionSupport 类的 getText 方法。

(3)表单元素的 Label 里: 为表单元素指定一个 key 属性

输出带占位符的国际化消息

Struts2 中提供了如下两种方式来填充消息字符串中的占位符

(1)JSP 页面, 在<s:text.../>标签中使用多个<s:param.../>标签来填充消息中的占位符。

(2)Action 中, 在调用 getText 方法时使用

getText(String aTextName,List args)或 getText(String key, String[] args)方

法来填充占位符。

除此之外，**Struts2** 还提供了对占位符的一种替代方式，这种方式允许在国际化消息资源文件中使用表达式，对于这种方式，则可避免在使用国际化消息时还需要为占位符传入参数值。如下在消息资源中使用表达式

succTip={\$username}, 欢迎, 您已经登录!

在上面的消息资源中，通过使用表达式，可以从 **ValueStack** 中取出该 **username** 属性值，自动填充到该消息资源中。

加载资源文件的方式

(1)加载全局资源文

件：
`<constant name="struts.custom.i18n.resources" value="baseName"/>`

(2)包范围资源文件：为 **Struts2** 指定包范围资源文件的方法是,在包的根路径下建立多个文件名为 **package_language_country.properties** 的文件，一旦建立了这个系列的国际化资源文件，应用中处于该包下的所有 **Action** 都可以访问该资源文件。需要注意的是上面的包范围资源文件的 **baseName** 就是 **package**，不是 **Action** 所在的包名。

(3)**Action** 范围资源文件：在 **Action** 类文件所在的路径建立多个文件名为 **ActionName_language_country.properties** 的文件。

(4)临时指定资源文件：`<s:i18n.../>`标签的 **name** 属性指定临时的国际化资源文件

对于在 **JSP** 中访问国际化消息，则简单的多，他们又可以分为两种形式：

(1)对于使用`<s:i18n.../>`标签作为父标签的`<s:text.../>`标签、表单标签的形式：

a、将从<s:i18n.../>标签指定的国际化资源文件中加载指定 **key** 对应的消息。

b、如果在 a 中找不到指定 **key** 对应的消息，则查找 **struts.custom.i18n.resources** 常量指定 **baseName** 的系列资源文件。

c、如果经过上面步骤一直找不到该 **key** 对应的消息，将直接输出该 **key** 的字符串值。

(2)如果<s:text.../>标签、表单标签没有使用<s:i18n.../>标签作为父标签：

直接加载 **struts.custom.i18n.resources** 常量指定 **baseName** 的系列资源文件。如果找不到该 **key** 对应的消息，将直接输出该 **key** 的字符串值。允许用户自行选择程序语言

Struts2 国际化的运行机制

在 Struts2 中，可以通过 **ActionContext.getContext().setLocale(Locale arg)** 设置用户的默认语言。为了简化设置用户默认语言环境，Struts2 提供了一个名为 **i18n** 的拦截器(Interceptor),并且将其注册在默认的拦截器中

(**defaultStack**)。 **i18n** 拦截器在执行 **Action** 方法前，自动查找请求中一个名为 **request_locale** 的参数。如果该参数存在，拦截器就将其作为参数，转换成 **Locale** 对象，并将其设为用户默认的 **Locale**(代表国家/语言环境)。除此之外，**i18n** 拦截器还会将上面生成的 **Locale** 对象保存在用户 **Session** 的名为 **WW_TRANS_I18N_LOCALE** 的属性中。一旦用户 **Session** 中存在一个名为 **WW_TRANS_I18N_LOCALE** 的属性，则该属性指定的 **Locale** 将会作为浏览者的默认 **Locale**。

[html] [view plaincopyprint?](#)

```

1. <%@ page language=      contentType=      %>
2. <%@taglib prefix=      uri=      %>
3. <script. type=      >
4. function langSelector_onChanged()
5. {
6. document.getElementById("langForm").submit();
7. }
8. </script>
9. <%-- 设置 SESSION_LOCALE 为用户 session 中的 WW_TRANS_I18N_LOCALE
    属性值 --%>
10. <s:set name=      value=
    />
11. <%-- 使用 lee.Locales 创建 locales 实例 --%>
12. <s:bean id=      name=      >
13. <%-- 为 locales 实例传入 current 参数值，如果 SESSION_LOCALE 为空，则返回
    ValueStack 中 locale 属性值(即用户浏览器设置的 Locale) --%>
14. <s:param name=      value=
    />
15. </s:bean>
16. <%-- 让用户选择语言的表单 --%>
17. <form. action=      id=
18. style=      >
19. <s:text name=      />
20. <s:select label=      list=      listKey=      listValue=
21. value=
22. name=      id=
23. nchange=      theme=      />
24. </form>

```

在其他页面中包含该页面:

[html] [view plaincopyprint?](#)

1. `<s:include value=` `/>`
2. 在 `struts.xml` 文件中增加 Action 通配符的配置:
3. `<?xml version=` `encoding=` `?>`
4. `<!DOCTYPE struts PUBLIC`
5. `"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"`
6. `"http://struts.apache.org/dtds/struts-2.0.dtd">`
7. `<struts>`
8. `<constant name=` `value=` `/>`
9. `<constant name=` `value=` `/>`
10. `<package name=` `extends=` `>`
11. `<!-- 使用通配符定义 Action 的 name -->`
12. `<action name=` `>`
13. `<!-- 将请求转发给/WEB-INF/jsp/路径下同名的 JSP 页面 -->`
14. `<result>/WEB-INF/jsp/{1}.jsp</result>`
15. `</action>`
16. `</package>`
17. `</struts>`

用户主动选择国际化应用介绍

首先配置 `struts.xml`

```
<constant name="struts.custom.i18n.resources" value="messageResource"></constant>
```

然后编写: `messageResource_zh_CN.properties` 和 `messageResource_en_US.properties`

具体代码示例:

`stuNumber=StudentNumber`

`password=Password`

`login=login`

`loginInfo=Play login`

语言选择的地方可以使用链接到 `action` 中, 需向 `action` 传递 `request_locale=en_US` 或者

`request_locale=zh_CN` 参数就可以简单的实现语言的切换

页面代码如下：

```
<a href="languageAction?request_locale=en_US">en</a>
```

```
<a href="languageAction?request_locale=zh_CN">cn</a>
```

然后再 *action* 中直接返回即可，在返回的界面得到 *messageResource_zh_CN* 的属性值，如下

代码示例：

```
label="%{getText('stuNumber')}"
```

```
label="%{getText('password')}"
```

（六十七）细谈 Spring（一）spring 简介

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许您选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。然而，Spring 的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何 Java 应用都可以从 Spring 中受益。Spring 的核心是个轻量级容器（container），实现了 IoC（Inversion of Control）模式的容器，Spring 的目标是实现一个全方位的整合框架，在 Spring 框架下实现多个子框架的组合，这些子框架之间彼此可以独立，也可以使用其它的框架方案加以替代，Spring 希望提供 one-stop shop 的框架整合方案

简单来说，Spring 是一个轻量级的控制反转(IOC)和面向切面（AOP）的容器框架。

轻量——从大小与开销两方面而言 Spring 都是轻量的。完整的 Spring 框架可以在一个大小只有 1MB 多的 JAR 文件里发布。并且 Spring 所需的处理开销也是微不足道的。此外，Spring 是非侵入式的：典型地，Spring 应用中的对象不依赖于 Spring 的特定类。

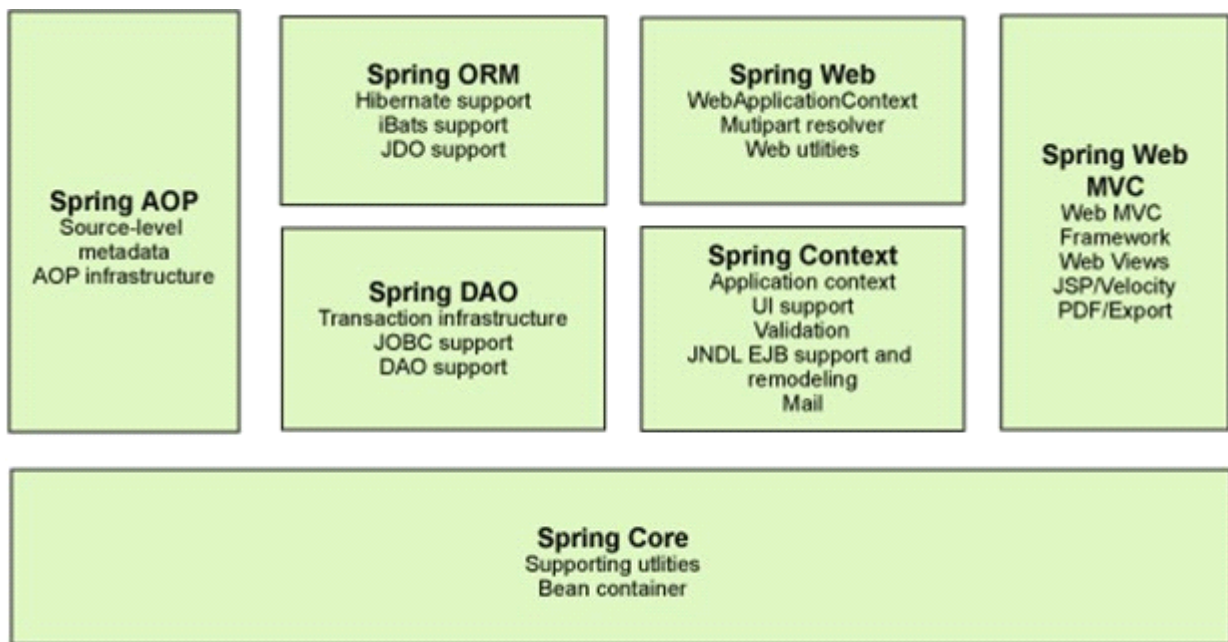
控制反转——Spring 通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了 IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为 IoC 与 JNDI 相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

面向切面——Spring 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（**auditing**）和事务（**transaction**）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。

容器——Spring 包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个 **bean** 如何被创建——基于一个可配置**原型**（**prototype**），你的 **bean** 可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，**Spring** 不应该被混同于传统的重量级的 **EJB** 容器，它们经常是庞大与笨重的，难以使用。

框架——Spring 可以将简单的组件配置、组合成为复杂的应用。在 **Spring** 中，应用对象被声明式地组合，典型地是在一个 **XML** 文件里。**Spring** 也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。所有 **Spring** 的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为 **Spring** 中的各种模块提供了基础支持。

Spring 框架是一个分层架构，由 7 个定义良好的模块组成。**Spring** 模块构建在核心容器之上，核心容器定义了创建、配置和管理 **bean** 的方式，如下图所示：



Spring 框架的分为 7 个模块，组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

1.核心容器：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 **BeanFactory**，它是工厂模式的实现。**BeanFactory** 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

2.Spring 上下文：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。

3.Spring AOP：通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序

中的对象提供了事务管理服务。通过使用 **Spring AOP**，不用依赖 **EJB** 组件，就可以将声明性事务管理集成到应用程序中。

4. Spring DAO: JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

5.Spring ORM: Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

6. Spring Web 模块: Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。

7. Spring MVC 框架: MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

Spring 框架的功能可以用在任何 J2EE 服务器中，大多数功能也适用于不受管理的环境。Spring 的核心要点是：支持不绑定到特定 J2EE 服务

的可重用业务和数据访问对象。毫无疑问，这样的对象可以在不同 J2EE 环境（Web 或 EJB）、独立应用程序、测试环境之间重用。

总结起来，Spring 有如下优点：

- 1.低侵入式设计，代码污染极低
2. 独立于各种应用服务器，可以真正实现 Write Once,Run Anywhere 的承诺
- 3.Spring 的 DI 机制降低了业务对象替换的复杂性
- 4.Spring 并不完全依赖于 Spring，开发者可自由选用 Spring 框架的部分或全部

IOC 和 AOP:这两个概念可以算是 spring 中的核心了，这两个概念非常抽象，并且也不容易理解，所以我们下面来简单来描述一下这两个概念。

控制反转模式（也称作依赖性介入）的基本概念是：不创建对象，但是描述创建它们的方式。在代码中不直接与对象和服务连接，但在配置文件中描述哪一个组件需要哪一项服务。容器（在 Spring 框架中是 IOC 容器）负责将这些联系在一起。在典型的 IOC 场景中，容器创建了所有对象，并设置必要的属性将它们连接在一起，决定什么时间调用方法。下表列出了 IOC 的一个实现模式。

类 服务需要实现专门的接口，通过接口，由对象提供这些服务，可

型 1 以从对象查询依赖性（例如，需要的附加服务），这种用的很少

类

通过 JavaBean 的属性（例如 setter 方法）分配依赖性

型 2

类

型 3

依赖性以构造函数的形式提供，不以 **JavaBean** 属性的形式公开

Spring 框架的 **IOC** 容器主要采用类型 2 和类型 3 实现。

面向方面的编程，即 **AOP**，是一种编程技术，它允许程序员对横切关注点或横切典型的职责分界线的行为（例如日志和事务管理）进行模块化。**AOP** 的核心构造是方面，它将那些影响多个类的行为封装到可重用的模块中。

AOP 和 **IOC** 是补充性的技术，它们都运用模块化方式解决企业应用程序开发中的复杂问题。在典型的面向对象开发方式中，可能要将日志记录语句放在所有方法和 **Java** 类中才能实现日志功能。在 **AOP** 方式中，可以反过来将日志服务模块化，并以声明的方式将它们应用到需要日志的组件上。当然，优势就是 **Java** 类不需要知道日志服务的存在，也不需要考虑相关的代码。所以，用 **Spring AOP** 编写的应用程序代码是松散耦合的。**AOP** 的功能完全集成到了 **Spring** 事务管理、日志和其他各种特性的上下文中。

IOC 容器：**Spring** 设计的核心是 `org.springframework.beans` 包，它的设计目标是与 **JavaBean** 组件一起使用。这个包通常不是由用户直接使用，而是由服务器将其用作其他多数功能的底层中介。下一个最高级抽象是 **BeanFactory** 接口，它是工厂设计模式的实现，允许通过名称创建和检索对象。**BeanFactory** 也可以管理对象之间的关系。

BeanFactory 支持两个对象模型:

1.单态 模型提供了具有特定名称的对象的共享实例，可以在查询时对其进行检索。**Singleton** 是默认的也是最常用的对象模型。对于无状态服务对象很理想。

2. 原型 模型确保每次检索都会创建单独的对象。在每个用户都需要自己的对象时，原型模型最适合。

bean 工厂的概念是 **Spring** 作为 **IOC** 容器的基础。**IOC** 将处理事情的责任从应用程序代码转移到框架。正如我将在下一个示例中演示的那样，

Spring 框架使用 **JavaBean** 属性和配置数据来指出必须设置的依赖关系。

Spring 的源码设计精妙、结构清晰、匠心独用，处处体现着大师对 **java** 设计模式灵活运用以及对 **Java** 技术的高深造诣。**Spring** 框架源码无疑是 **Java** 技术的最佳实践范例。如果想在短时间内迅速提高自己的 **Java** 技术水平和应用开发水平，学习和研究 **Spring** 源码将会使你收到意想不到的效果。可是从哪着手研究 **Spring** 却是很多新手头疼的地方，下面的参考资料将带领大家慢慢的深入解析 **Spring**

1 **Spring** 中的事务处理

2 **ioc** 容器在 **Web** 容器中的启动

3 **Spring JDBC**

4 **Spring MVC**

5 **Spring AOP** 获取 **Proxy**

6 **Spring** 声明式事务处理

7 **Spring AOP** 中对拦截器调用的实现

8 Spring 驱动 Hibernate 的实现

9 Spring Acegi 框架鉴权的实现

具体的源码解析内容大家可以百度或者谷歌一下，可以找到大量源码解析内容。

（六十八）细谈 Spring（二）自己动手模拟 spring

在我们学习 spring 之前，根据 spring 的特性，我们来自己来模拟一个 spring 出来，也就是说不利用 spring 来实现 spring 的效果。本实例主要是实现 spring 的 IOC 功能。

点击下载源码：用力点

首先我们把我们要用的 dao、service、entity 定义出来：

Student.java :

[java] [view plaincopyprint?](#)

```
1. package com.bzu.entity;
2. public class Student {
3.     private int id;
4.     private String name;
5.     private String address;
6.     *****set、get 方法省略
7. }
```

因为 spring 提倡的就是面向接口编程，所以在我们写 dao 层和 service 层具体实现之前，我们先定义接口，让我们的具体实现实现接口。接口的代码很简单，在这就不贴出来了。

[java] [view plaincopyprint?](#)

```
1. StudentdaoImp.java
2. public class StudentDaoImp implements StudentDao {
3.     public void add(Student stu) {
4.         System.out.println("stu is saved");
```

```
5. }  
6. }
```

StudentServiceImpl.java

[java] [view plaincopyprint?](#)

```
1. public class StudentServiceImpl implements StudentService {  
2.     StudentDao stuDao=null;  
3.     public StudentDao getStuDao() {  
4.         return stuDao;  
5.     }  
6.     public void setStuDao(StudentDao stuDao) {  
7.         this.stuDao = stuDao;  
8.     }  
9.     @Override  
10.    public void add(Student stu) {  
11.        stuDao.add(stu);  
12.    }  
13.}
```

这里要注意的是，我们这里是模拟 spring，主要模拟 spring 中的 IOC 功能，所以在此我们一样要在 service 层中定义 dao 的实例，当然不用 new 出来，我们就通过 spring 的 IOC 把这里的 dao 层注入进来。不要忘了对 dao 提供 set。Get 方法，因为 IOC 的底层其实就是利用反射机制实现的，他把 dao 注入进来，其实底层就是通过反射 set 进来的。

好了，我们所需的 dao 层、service 层还有 entity 定义好了之后，万事俱备只欠东风了，下一步我们就是定义我们自己的

ClassPathXmlApplicationContext 类了，通过他，在我们 new 出他的对象的时候，他来加载配置文件，然后把我们的 dao 操作注入到我们的 service

层,在 spring 中, `ClassPathXmlApplicationContext` 类实现了 `BeanFactory` 接口,在此我们也定义一个 `BeanFactory` 接口,其实这个接口没什么具体的作用,我们就是为了来模拟 spring。在定义这个接口和实现类之前,我们先来看一下我们所需的 xml 是怎么编写的,下面我们就具体来看一下 `beans.xml` 的配置:

Beans.xml:

[\[html\] view plaincopyprint?](#)

```
1. <beans>
2. <bean id=          class=
3. <bean id=          class=
4. <property name=      bean=
5. </bean>
6. </beans>
```

看到这,相信大家都能感觉到这个配置文件太简单了,没有 spring 中那么多繁琐的配置,当然啦,我们这是只是实现其中的一个功能, spring 提供了很多那么强大的功能,配置当然繁琐一些了。相信上边的代码不用我解释大家也能看懂了吧。

好了,配置文件我们看完了,下一步我们一起来看一下我们的 spring 容器——`ClassPathXmlApplicationContext` 具体是怎么实现的,我们首先还是来看一下他的接口定义:

BeanFactory.java:

[\[java\] view plaincopyprint?](#)

```
1. public interface BeanFactory {
2.     public Object getBean(String id);
```

3. }

我们看到，接口其实很简单，就定义了一个 `getBean` 方法，下面我们来看一下具体的实现类：

ClassPathXmlApplicationContext.java

[java] [view plain](#) [copy](#) [print?](#)

```
1. public class ClassPathXmlApplicationContext implements BeanFactory {
2.     private Map<String, Object> beans = new HashMap<String, Object>();
3.     public ClassPathXmlApplicationContext() throws Exception, Exception {
4.         SAXBuilder sb = new SAXBuilder();
5.         Document doc = sb.build(this.getClass().getClassLoader()
6.             .getResourceAsStream("beans.xml")); // 构造文档对象
7.         Element root = doc.getRootElement(); // 获取根元素 HD
8.         List list = root.getChildren("bean");// 取名字为 disk 的所有元素
9.         for (int i = 0; i < list.size(); i++) {
10.            Element element = (Element) list.get(i);
11.            String id = element.getAttributeValue("id");
12.            String clazz = element.getAttributeValue("class");
13.            Object o = Class.forName(clazz).newInstance();
14.            System.out.println(id);
15.            System.out.println(clazz);
16.            beans.put(id, o);
17.            for (Element propertyElement : (List<Element>) element
18.                .getChildren("property")) {
19.                String name = propertyElement.getAttributeValue("name"); // userDAO
20.                String bean = propertyElement.getAttributeValue("bean"); // u
21.                Object beanObject = beans.get(bean);// UserDaoImpl instance
22.                String methodName = "set" + name.substring(0, 1).toUpperCase()
23.                    + name.substring(1);
24.                System.out.println("method name = " + methodName);
25.                Method m = o.getClass().getMethod(methodName,
26.                    beanObject.getClass().getInterfaces()[0]);
```

```
27.m.invoke(o, beanObject);
28.}
29.}
30.}
31.@Override
32.public Object getBean(String id) {
33.return beans.get(id);
34.}
35.}
```

代码贴出来了，不知道大家看懂没有。下面我来解释一下这段代码：

首先我们定义了一个容器 `Map<String, Object> beans`，这个容器的作用就是用来装我们从配置文件里解析来的一个个 `bean`，为什么要用 `map` 类型，我想大家也差不多能猜到吧，我们配置文件中每一个 `bean` 都有一个 `id` 来作为自己的唯一身份。我们把这个 `id` 存到 `map` 的 `key` 里面，然后 `value` 就装我们的具体 `bean` 对象。说完这个容器之后，下面我们在来看一下

ClassPathXmlApplicationContext 的构造方法，这个构造方法是我们 **spring** 管理容器的核心，这个构造方法的前半部分是利用的 `jdom` 解析方式，把 `xml` 里面的 `bean` 一个个的解析出来，然后把解析出来的 `bean` 在放到我们 `bean` 容器里。如果这段代码看不懂的话，那你只好在去看看 `jdom` 解析 `xml` 了。好了，我们下面在来看一下这个构造的方法，后半部分主要是在对配置文件进行解析出 `bean` 的同时去查看一下这个 `bean` 中有没有需要注射 `bean` 的，如果有的话，他就去通过这些里面的 `property` 属性获取他要注射的 `bean` 名字，然后构造出 `set` 方法，然后通过反射，调用注入 `bean` 的 `set` 方法，这样我们所需要的 `bean` 就被注入进来了。如果这段代码你看不懂的话，那你

只能去看一下有关反射的知识了。最后我们就来看一下实现接口的 `getBean` 放了，其实这个方法很简单，就是根据提供的 `bean` 的 `id`，从 `bean` 容器内把对应的 `bean` 取出来。

好了，我们所需的东西都定义好了，下面我们据来测试一下，看看我们自己模仿的 `spring` 到底能不能自动把我们所需要的 `dao` 层给我们注入进来。

[java] [view plaincopyprint?](#)

```
1. public static void main(String[] args) throws Exception {
2.   ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext()
   ;
3.   Student stu = new Student();
4.   StudentService service = (StudentService) context.getBean("stuService");
5.   service.add(stu);
6. }
```

运行代码，控制台输出：

```
com.bzu.service.imp.StudentServiceImp
method name = setStuDao
stu is saved
```

好，成功注入进来，到此，我们模仿的 `spring` 就到此结束了，下一篇我们就开始对 `spring` 进行一个全面深入了解了，敬请期待。

（六十九）细谈 Hibernate（十七）Hibernate 实现分页和综合查询 详解

现如今，在 web 系统项目中，分页及综合查询几乎成了不可缺少的功能，每一个实体列表几乎都要要求带有分页及综合查询，前几天做老师布置的作业，想着干脆做一个通用点的，省得以后再每一次都要写一遍了。下面我们就一起来看一下我用 hibernate 实现的通用分页及综合查询。当然我这里所属的通用并不似绝对的，每到一个不同的场合，前台页面和数据接收还是稍微的要改一下：

首先我们先来看一下列表的 jsp 页面：list.jsp：

[html] [view plain](#) [copy](#) [print?](#)

```
1. <SPAN style=                                ><body>
2. <!-- 综合查询操作层 -->
3. <div id=                                >
4. <s:form action=                                theme=                                target=                                >
5. <fieldset style=                                >
6. <legend>查询条件</legend>
7. <s:hidden name=                                value=                                ></s:hidden>
8. 课程名:<s:textfield name=                                cssClass=                                />
9. 课程号:<s:textfield name=                                cssClass=                                />
10. 开课学期:<s:textfield name=                                cssClass=                                />
11. 课时: 从<s:textfield name=                                cssClass=                                />至
    <s:textfield name=                                cssClass=                                />课时<br/>
12. 课程学分:<s:textfield name=                                cssClass=                                />
13. <s:submit value=                                cssClass=                                />
14. </fieldset>
15. </s:form>
16. </div>
17. <table class=                                >
18. <tr>
```

```

19. <th colspan=      >操作</th>
20. <th>课程号</th>
21. <th>课程名</th>
22. <th>开课学期</th>
23. <th>课时</th>
24. <th>学分</th>
25. </tr>
26. <c:forEach items=      var=      varStatus=      >
27. <tr bgColor=      >
28. <td><img src=      alt=
    onclick=      /></td>
29. <td><a href=      ><img src=
    alt=      /></a></td>
30. <td>${cour.id}</td>
31. <td>${cour.name }</td>
32. <td>${cour.lessonTime }</td>
33. <td>${cour.lessonHour}</td>
34. <td>${cour.lessonPoints }</td>
35. </tr>
36. </c:forEach>
37. </table>
38. <br/>
39. <center>
40. <div id=      >
41. <c:set var=      value=      />
42. <fmt:formatNumber var=      value=      pattern=      />
43. <ul>
44. <li style=      >
45. 第${sessionScope.thisindex }/${lastIndex }页
46. </li>
47. <li style=      >
48. <a href=      target=      >首页</a>
49. <!--
50. <c:set var="pageCount" value="${fn:length(userList)%10==0?fn:length(userList)/1
    0:fn:length(userList)/10+1 }"/>

```

```

51.-->
52.  <c:forEach var=    begin=    step=    end=    >
53. <a href=            target=            ><c:out value
    =    /></a>
54.</c:forEach>
55.<a href=            target=            >尾页
    </a>
56.</li>
57. <li style=            >
58.<s:form action=            theme=            target=            >
59.第<s:textfield name=            cssStyle=            />页
60.<s:submit value=            id=            />
61.</s:form>
62.</li>
63.</ul>
64.</div>
65.</center>
66.</body></SPAN>

```

这个列表的 jsp 页面下面的分页层差不多可以说是通用的吧，直接修改一下 action 的名字就可以了。上面数据列表要因不同的数据列表而不同了。我们来看一下他的分页层，首先看一下

`<c:set var="pageCount" value="${(coucount-1)/10+1 }"/>`，根据在 action 中保存的课程数目来获得总共的页数。这里的页数也就是最后一页的页数。然后获得在 action 中保存的当前是第几页。接下来就是首页、尾页以及页数列表的链接了。并且还有一个可以输入页数进行查询的。

最上面是综合查询层，这个层其实很简单，就是一个 form 表单，根据输入的内容进行查询。

好了具体的页面看完了，下一步我们来看一下在 action 中接收数据的的地方，是怎么接收数据的。

```
public class CourseListAction extends ActionSupport {
private final int EVpAGECOUNT = 10;
private int index;
private Course course;
private int startHour;
private int endHour;
```

*****省略 set。Get 方法

```
public String execute() {
    String hql = "";
    if (course != null) {
        if (course.getName() != null && !("").equals(course.getName()))
            hql += " and s.name like '%" + course.getName() + "%'";
        if (course.getId() != 0)
            hql += " and s.id =" + course.getId();
        if (course.getLessonTime() != 0)
            hql += " and s.lessonTime =" + course.getLessonTime();
        if (course.getLessonPoints() != 0)
            hql += " and s.lessonPoints =" + course.getLessonPoints();
        if (startHour != 0 && endHour != 0 && !("").equals(startHour)
            && !("").equals(endHour))
            hql += " and s.lessonHour between '" + startHour + "' and '"
                + endHour + "'";
    }
    List<Object> list = index != 0 ? dao.pageList(index, Course.class, hql)
        : dao.pageList(1, Course.class, hql);

    HttpSession session = ServletActionContext.getRequest().getSession();
    session.setAttribute("list", list);
    session.setAttribute("thisindex", index == 0 ? 1 : index);

    Session session1 = HibernateUtil.getSession();
    session1.beginTransaction();
    session.setAttribute("coucount", session1.createQuery("from Course")
        .list().size());
    System.out.println(session1.createQuery("from Course").list().size()
        + ".....");
    if (flag != 0)
        return "front";
    ---
}
```

下面我们来解释一下这个 action，首先来看一下它所定义的变量，

index 变量其实就是当前要查看的第几页，Course 变量的是接收前面综合查

询传递的数据的 **starthour** 和 **endHour** 其实是前面综合查询的生日的起始时间和终止时间。我们在来看一下一个常量，**EVpAGECOUNT** 这个常量是定义每页的数据量的。说完变量我们在大体看一下 **execute** 方法里面的内容：首先前面是根据综合查询的每一项是否为空来组装这里的 hql 语句，很简单，不说了，然后就是根据查询的对象、当前页和 hql 语句进行综合和分页查询了。最后就是保存一些数据了。好了，最后我们来看一下 dao 层所写的通用分页和综合查询的一个方法：











```
public static List<Object> selectPage(int index, Class clazz, String bhql) {  
  
    final int PAGETOTAL = 10;  
    Session session = null;  
    Transaction tran = null;  
    List<Object> list = null;  
  
    try {  
        session = HibernateUtil.getSession();  
        tran = session.beginTransaction();  
        System.out.println(clazz.getName());  
        String hql = "from " + clazz.getName() + " as s where 1=1" + bhql;  
        System.out.println(hql);  
        Query query = session.createQuery(hql);  
        query.setFirstResult((index - 1) * PAGETOTAL);  
        query.setMaxResults(PAGETOTAL);  
        list = query.list();  
        tran.commit();  
    } catch (Exception e) {  
        tran.rollback();  
        e.printStackTrace();  
    }  
    return list;  
}
```

这个方法应该是对所有的分页和综合查询都是通用的，应该不需要改任何代码，把相应的值传过去就可以了。

运行效果图：

查询条件

课程名: 课程号: 开课学期: 课时: 从 课程学分:

操作		课程号	课程名	开课
	<input type="button" value="x"/>	1	java	4
	<input type="button" value="x"/>	2	c#	1
	<input type="button" value="x"/>	4	java	1
	<input type="button" value="x"/>	19	计算机文化基础	111
	<input type="button" value="x"/>	20	Flash	2
	<input type="button" value="x"/>	21	photoshop	2
	<input type="button" value="x"/>	22	高数	22
	<input type="button" value="x"/>	23	英语	2
	<input type="button" value="x"/>	24	.net高级	2
	<input type="button" value="x"/>	25	语文	2

第1/2页 首页 1 2 尾页 第 页

（七十）细谈 Spring（三）IOC 和 spring 基本配置详解

对于 IoC 的一些知识点，相信大家都知道他在 Spring 框架中所占有的地位，应该可以算的上是核心之一吧，所以 IOC 是否理解清楚，决定了大家对 Spring 整个框架的理解

IoC 的理解

spring 的两个核心概念：一个是控制反转 IoC，也可以叫做依赖注入 DI。还有一个是面向切面编程 AOP。

控制反转：当某个 java 对象需要（依赖）另一个 java 对象时，不是自身直接创建依赖对象，而是由实现 IoC 的容器（如 spring 框架的 IoC 容器）来创建，并将它注入需要这个依赖对象的 java 对象中。

spring 的容器

spring 管理的基本单元是 Bean，在 spring 的应用中，所有的组件都是一个一个的 Bean，它可以是任何的 java 对象。spring 负责创建这些 Bean 的实例。并管理生命周期。而 spring 框架是通过其内置的容器来完成 Bean 的管理的，Bean 在 spring 的容器中生存着，使用时只需要通过它提供的一些方法从其中获取即可。

spring 的容器有两个接口：BeanFactory 和 ApplicationContext 这两个接口的实例被陈为 spring 的上下文。

[Java] [view plaincopyprint?](#)

```
1. <SPAN style="FONT-SIZE: 18px">ApplicationContext ac = new ClassPathXmlApplicationContext("app*.xml");
```

```
2. AccountService accountService =(AccountService)ac.getBean("accountServiceIm  
pl");</SPAN>
```

注：由于 **ApplicationContext** 是基于 **BeanFactory** 之上的，所以，一般 **ApplicationContext** 功能比较强大，建议使用

ApplicationContext 经常用到的三个实现：

1.**ClassPathXmlApplicationContext**：从类路径中的 XML 文件载入上下文定义信息。把上下文定义文件当成类路径资源。

2.**FileSystemXmlApplicationContext**：从文件系统中的 XML 文件载入上下文定义信息。

3.**XmlWebApplicationContext**：从 Web 系统中的 XML 文件载入上下文定义信息。

一：spring 的依赖注入

1)、构造器注入

[html] view plaincopyprint?

```
1. <SPAN style=                ><bean id=                class=  
  
2. scope="singleton"/>  
3. <bean id=  
4. class=                scope="">  
5. <!-- 构造方法注入方式-->  
6. <constructor-arg ref=                />  
7. </bean></SPAN>
```


注：这种注入方式很少用，如果是注入对象一般为上例注入，但有时要注入基本数据类型，一般用下面方法注入

[html] view plaincopyprint?

1. `<constructor-arg>`
2. `<value>hello world!</value>`
3. `</constructor-arg>`

如果构造方法不只一个参数时，应指明所注入参数的索引或者数据类型，例如：

[html] view plaincopyprint?

1. `<constructor-arg index="0" type="java.lang.String">`
2. `<value>sunDriver</value>`
3. `</constructor-arg>`
4. `<constructor-arg index="1" type="java.lang.String">`
5. `<value>jdbc:odbc:School</value>`
6. `</constructor-arg>`

2)、设值（set 方法）注入

[html] view plaincopyprint?

1. `<bean id="accountDaoImpl" class="com.dao.impl.AccountDaoImpl">`
2. `class="com.dao.impl.AccountDaoImpl">`
3. `<bean id="accountDaoImpl" class="com.dao.impl.AccountDaoImpl">`
4. `class="com.dao.impl.AccountDaoImpl">`
5. `<!-- 设值（set 方法）注入-->`
6. `<property name="accountDaoImpl" ref="accountDaoImpl"/>`
7. `</bean>`

注：`<property name="accountDaoImpl" ref="accoutDaoImpl"/>`

相当于调用 `set AccountDaoImpl` 方法，把值设为 `accoutDaoImpl`

3) 接口注入（很少用）

二：xml 装配 Bean 属性含义

1.id:指定该 Bean 的唯一标识。

2.class:指定该 Bean 的全限定名。

3.name:为该 Bean 指定一到多个别名。多个别名可以用“，”和“；”分割。

4.autowire:指定该 Bean 的属性的装配方式。

所谓自动装配是指在<BEAN>标签中不用指定其依赖的 BEAN，而是通过配置的自动装配来自动注入依赖的 BEAN,这种做法让我们的配置更加简单

1) no:不使用自动装配。必须通过 `ref` 元素指定依赖，这是默认设置。由于显式指定协作者可以使配置更灵活、更清晰，因此对于较大的部署配置，推荐采用该设置。而且在某种程度上，它也是系统架构的一种文档形式。

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                                ><bean id=                class=
2. scope=                                     >
3. <property name=
4. ref=                                     >
5. </property>
6. </bean></SPAN>
```

备注：有 `property` 属性指定 `ref`

2) `byName`:根据属性名自动装配。此选项将检查容器并根据

名字查找与属性完全一致的 `bean`，并将其与属性自动装配。例如，在

bean 定义中将 `autowire` 设置为 `by name`，而该 bean 包含 `master` 属性（同时提供 `setMaster(..)` 方法），Spring 就会查找名为 `master` 的 bean 定义，并用它来装配给 `master` 属性。

```
<bean id="bean1" class="cn.csdn.service.Bean1"
scope="singleton" autowire="byName"/>
```

备注：没有 `property` 属性

3) `byType`:如果容器中存在一个与指定属性类型相同的

bean，那么将与该属性自动装配。如果存在多个该类型的 bean，那么将会抛出异常，并指出不能使用 `byType` 方式进行自动装配。若没有找到相匹配的 bean，则什么事都不发生，属性也不会被设置。如果你不希望这样，那么可以通过设置 `dependency-check="objects"` 让 Spring 抛出异常。

备注：spring3.0 以上不抛异常。

```
<bean id="bean1" class="cn.csdn.service.Bean1"
scope="singleton" autowire="byType"/>
```

备注：没有 `property` 属性

4) `Constructor`:与 `byType` 的方式类似，不同之处在于它应用

于构造器参数。如果在容器中没有找到与构造器参数类型一致的 bean，那么将会抛出异常。

```
<bean id="bean1" class="cn.csdn.service.Bean1"
scope="singleton" autowire="constructor"/>
```

备注：没有 `property` 属性

5) `autodetect`:通过 bean 类的自省机制（`introspection`）来决定是使用

`constructor` 还是 `byType` 方式进行自动装配。如果发现默认的构造器，那么将使用 `byType` 方式。

```
<bean id="bean1" class="cn.csdn.service.Bean1"
scope="singleton" autowire="autodetect"/>
```

5.scope:指定该 Bean 的生存范围

scope 用来声明 IOC 容器中的对象应该处的限定场景或者说该对象的存活空间，即在 IOC 容器在对象进入相应的 scope 之前，生成并装配这些对象，在该对象不再处于这些 scope 的限定之后，容器通常会销毁这些对象。

1) **singleton** 类型的 bean 定义，在一个容器中只存在一个实例，所有对该类型 bean 的依赖都引用这一单一实例

2) **scope** 为 prototype 的 bean，容器在接受到该类型的对象的请求的时候，会每次都重新生成一个新的对象给请求方，虽然这种类型的对象的实例化以及属性设置等工作都是由容器负责的，但是只要准备完毕，并且对象实例返回给请求方之后，容器就不再拥有当前对象的引用，请求方需要自己负责当前对象后继生命周期的管理工作，包括该对象的销毁

3) **request** , **session** 和 **global session**

这三个类型是 spring2.0 之后新增的，他们只适用于 web 程序，通常是和 XmlWebApplicationContext 共同使用

request:

```
<bean id ="requestPrecessor" class="...RequestPrecessor" scope="request" />
```

Spring 容器，即 XmlWebApplicationContext 会为每个 HTTP 请求创建一个全新的 RequestPrecessor 对象，当请求结束后，该对象的生命周期即告结束

session

```
<bean id ="userPreferences" class="...UserPreferences" scope="session" />
```

Spring 容器会为每个独立的 session 创建属于自己的全新的 UserPreferences 实例，他比 request scope 的 bean 会存活更长的时间，其他的方面真是没什么区别。

global session:

```
<bean id="userPreferences" class="...UserPreferences" scope="globalsession" />
```

global session 只有应用在基于 portlet 的 web 应用程序中才有意义，他映射到 portlet 的 global 范围的 session，如果普通的 servlet 的 web 应用中使用了这个 scope，容器会把它作为普通的 session 的 scope 对待。

6.init-method:指定该 Bean 的初始化方法。destroy-method:指定该 Bean 的销毁方法。这个就像 servlet 中 init 和 destroy 方法一样，只不过这里在配置文件配置的

7.abstract:指定该 Bean 是否为抽象的。如果是抽象的，则 spring 不为它创建实例。

8.parent

如果两个 Bean 的属性装配信息很相似，那么可以利用继承来减少重复的配置工作。

<!-- 装配 Bean 的继承

父类作为模板，不需要实例化，设置 abstract="true"-->

```
` <bean id="parent" class="cn.csdn.service.Parent"
abstract="true">
<property name="name" value="z_xiaofei168"/>
<property name="pass" value="z_xiaofei168"/>
</bean>
```

<!-- 装配 Bean 的继承

子类中用 **parent** 属性指定父类标识或别名

子类可以覆盖父类的属性装配，也可以新增自己的属性装配

-->

```
<bean id="child" class="cn.csdn.service.Chlid"
parent="parent">
<property name="pass" value="123123"/>
<property name="age" value="22"/>
</bean>
```

三：装配 **Bean** 的各种类型属性值

1..简单类型属性值的装配

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                ><bean id=                class=
    >
2. <property name=                value=                />
3. <property name=                >
4. <value>22</value>
5. </property>
6. </bean></SPAN>
```

2.引用其他 **Bean** 的装配

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                ><bean id=                class=
    >
2. ...
3. </bean>
4. <bean id=                class=                >
5. <!-- 引用自其他 Bean 的装配-->
6. <property name=                ref=                />
7. </bean></SPAN>
```

另外一种不常使用的配置方式是在 `property` 元素中嵌入

一个 `bean` 元素来指定所引用的 Bean.

[html] [view plaincopyprint?](#)

```
1. <SPAN style=                ><bean id=                class=
    >
2. ...
3. </bean>
4. <bean id=                class=                >
5. <!-- 引用自其他 Bean 的装配-->
6. <property name=                >
7. <bean id=
8. class=                />
9. </property>
10.</bean></SPAN>
```

3.集合的装配

其实集合的装配并不是复杂，反而感觉到很简单，用一个例子来说明问题吧：

[java] [view plaincopyprint?](#)

```
1. <SPAN style="FONT-SIZE: 18px">package com.bebig.dao.impl;
2. import java.util.List;
3. import java.util.Map;
4. import java.util.Properties;
5. import java.util.Set;
6. import com.bebig.dao.UserDAO;
7. import com.bebig.model.User;
8. public class UserDAOImpl implements UserDAO {
9.     private Set<String> sets;
10.    private List<String> lists;
11.    private Map<String, String> maps;
12.    private Properties props;
13.    public Set<String> getSets() {
```

```

14.     return sets;
15. }
16. public void setSets(Set<String> sets) {
17.     this.sets = sets; }
18. public List<String> getLists() {
19.     return lists; }
20. public void setLists(List<String> lists) {
21.     this.lists = lists; }
22. public Map<String, String> getMaps() {
23.     return maps;
24. }
25. public void setMaps(Map<String, String> maps) {
26.     this.maps = maps;
27. }
28. public Properties getProps() {
29.     return props;
30. }
31. public void setProps(Properties props) {
32.     this.props = props;
33. }
34. public void save(User u) {
35.     System.out.println("a user saved!");
36. }
37. @Override
38. public String toString() {
39.     return "sets.size:" + sets.size() + " lists.size:" + lists.size()
40.         + " maps.size:" + maps.size() + " props.size:" + props.size();
41. }
42.}
43.</SPAN>

```

配置如下：

[html] [view plaincopyprint?](#)


```

1. <SPAN style=                                ><?xml version=      encoding=      ?>
2. <beans xmlns=
3.      xmlns:xsi=                                xmlns:context=

4.      >
5.      <!-- a service object; we will be profiling its methods -->
6.      <bean name=      class=                                >
7.          <!-- set -->
8.          <property name=      >
9.              <set>
10.                 <value>1</value>
11.                 <value>2</value>
12.             </set>
13.          </property>
14.          <!-- list -->
15.          <property name=      >
16.              <list>
17.                 <value>a</value>
18.                 <value>b</value>
19.             </list>
20.          </property>
21.          <!-- map -->
22.          <property name=      >
23.              <map>
24.                 <entry key=      value=      ></entry>
25.                 <entry key=      value=      ></entry>
26.             </map>
27.          </property>
28.          <!-- properties -->
29.          <property name=      >
30.              <props>
31.                 <prop key=      >haha</prop>
32.                 <prop key=      >hi</prop>
33.             </props>
34.          </property>

```

```
35. </bean>
36. <bean id=          class=
37.     scope=        >
38.     <constructor-arg>
39.         <ref bean=  />
40.     </constructor-arg>
41. </bean>
42. <!-- this switches on the load-time weaving -->
43. <!-- <context:load-time-weaver /> -->
44. </beans></SPAN>
```

四：Spring bean 生命周期

在传统的 Java 应用中，Bean 的生命周期非常简单。Java 的关键词 new 用来实例化 Bean（或许他是非序列化的）。这样就够用了。相反，Bean 的生命周期在 Spring 容器中更加细致。理解 Spring Bean 的生命周期非常重要，因为你或许要利用 Spring 提供的机会来订制 Bean 的创建过程。

bean 生命周期

- 1.容器寻找 Bean 的定义信息并且将其实例化。
- 2.受用依赖注入，Spring 按照 Bean 定义信息配置 Bean 的所有属性。
- 3.如果 Bean 实现了 BeanNameAware 接口，工厂调用 Bean 的 setBeanName() 方法传递 Bean 的 ID。
- 4.如果 Bean 实现了 BeanFactoryAware 接口，工厂调用 setBeanFactory()方法传入工厂自身。
- 5.如果 BeanPostProcessor 和 Bean 关联，那么它们的 postProcessBeforeInitialization()方法将被调用。

6.如果 Bean 指定了 init-method 方法，它将被调用。

7.最后，如果有 BeanPostProcessor 和 Bean 关联，那么它们的 postProcessAfterInitialization()方法将被调用。

到这个时候，Bean 已经可以被应用系统使用了，并且将被保留在 Bean Factory 中知道它不再需要。

有两种方法可以把它从 Bean Factory 中删除掉。

1.如果 Bean 实现了 DisposableBean 接口，destroy()方法被调用。

2.如果指定了订制的销毁方法，就调用这个方法。

Bean 在 Spring 应用上下文的生命周期与在 Bean 工厂中的生命周期只有一点不同，唯一不同的是，如果 Bean 实现了 ApplicationContextAware 接口，setApplicationContext()方法被调用。

只有 singleton 行为的 bean 接受容器管理生命周期。

non-singleton 行为的 bean，Spring 容器仅仅是 new 的替代，容器只负责创建。

对于 singleton bean，Spring 容器知道 bean 何时实例化结束，何时销毁，Spring 可以管理实例化结束之后，和销毁之前的行为，管理 bean 的生命周期行为主要未如下两个时机：

Bean 全部依赖注入之后

Bean 即将销毁之前

1) 依赖关系注入后的行为实现：

有两种方法：A.编写 init 方法 B.实现 InitializingBean 接口

afterPropertiesSet 和 init 同时出现，前者先于后者执行，使用 init 方法，需要对配置文件加入 init-method 属性

2) bean 销毁之前的行为

有两种方法：A.编写 close 方法 B.实现 DisposableBean 接口

destroy 和 close 同时出现，前者先于后者执行，使用 close 方法，需要对配置文件加入 destroy-method 属性

总体上分只四个阶段

1. BeanFactoryPostProcessor 实例化

2. Bean 实例化，然后通过某些 BeanFactoryPostProcessor 来进行依赖注入

3. BeanPostProcessor 的调用.Spring 内置的 BeanPostProcessor 负责调用 Bean 实现的接

口：BeanNameAware, BeanFactoryAware, ApplicationContextAware 等等，等这些内置的 BeanPostProcessor 调用完后才会调用自己配置的 BeanPostProcessor

4.Bean 销毁阶段

注：xml 依赖注入中的 bean.xml 例子：

[\[html\] view plaincopyprint?](#)

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
5.         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
6.     <bean id="..." class="...">
```

```
7. </bean>
8. <bean id=          class=          >
9.   <!--
10.   <property name="userDAO" ref="u" />
11.   -->
12.   <constructor-arg>
13.     <ref bean=    />
14.   </constructor-arg>
15. </bean>
16.</beans></SPAN>
```

（七十一）细谈 struts2 （十三）struts2 实现文件上传和下载详解

文件上传和文件下载是我们在 web 应用程序中常用的两个功能，在 java 中，实现这两种功能的方式也有很多种，其中 struts2 就给我们提供了一种算是比较简单的方式吧，下面我们就一起来看一下，首先我们来看文件上传：

文件上传

文件上传我们首先应该注意的是在上传页面的表单，这个表单也是有讲究的，由于我们提交表单的数据中有文件上传，所以这个表单的所使用的编码类型就不能是原来的了，在这里我们应该使用的编码方式是 `multipart/form-data`，并且数据提交方式要用 `post` 方式，下面我们具体来看一下：

Form.jsp:

[html] view plaincopyprint?

```
1. <form action=                target=                onsubmit=
    method=
2. enctype=                >
3. <table class=    width=                >
4. <td>姓名</td>
5. <td><input type=    name=                value=                /></td>
    >
6. </tr>
7. <tr bgColor=                >
8. <td>上传头像</td>
9. <td><input type=    name=                />
10.    </td>
11.</tr>
```

```

12.<tr bgColor=      >
13.<td colspan=  ><input type=      value=
    class=      /> <input type=      value=      class=      /></td>
14.</tr>
15.</table>
16.</form>

```

OK, 看完表单以后我们就要来看一下 `action` 里面是怎么来接收这些数据的, 其实也很简单, 直接在 `action` 中定义三个变量, 这三个变量分别是文件、文件名, 还有文件类型, 如下:

[java] `view plaincopyprint?`

```

1. private File file;
2. private String fileFileName;
3. private String fileContentType;

```

这三个变量的名字是有讲究的, 不是随便命名就 OK 了, 其中 `file` 这个变量名要和表单中文件的 `name` 要相同, `fileFileName` 这个也是固定的, 起名格式就是 `name+FileName`, 同样 `fileContentType` 也是如此, 命名规则是 `name+ContentType`, 只有你按照命名规则来定义变量, **struts2** 才能把文件上传相关信息收集起来。Ok, 看完了变量设置, 下面我们来看一下怎么 **struts2** 是怎么把文件上传到我们指定的位置的。那我们就先上代码, 让代码帮我们来理解:

[java] `view plaincopyprint?`

```

1. String root = ServletActionContext.getRequest().getRealPath("/upload");
2. try{
3. InputStream is = new FileInputStream(file);

```

```

4. // 创建一个文件，路径为 root，文件名叫 fileFileName
5. //自定义文件
   名 fileFileName="111"+fileFileName.substring(fileFileName.lastIndexOf("."));
6. File destFile = new File(root, fileFileName);
7. // 开始上传
8. OutputStream os = new FileOutputStream(destFile);
9. byte[] buffer = new byte[50000];
10.int length = 0;
11.// enctype="multipart/form-data"
12.while (-1 != (length = is.read(buffer))) {
13.os.write(buffer, 0, length);
14.}

```

我们可以看到，就是简单的几行代码，其实并不难，他主要就是利用了 IO 流来实现的文件上传。单文件上传实现以后，多文件上传实现起来就不难了。

多文件上传

与单文件上传相似，Struts 2 实现多文件上传也很简单。你可以使用多个 `<s:file />` 绑定 Action 的数组或列表。

[html] view plaincopyprint?

```

1. <s:form action =                method =                enctype =
   >
2. <s:file label =                name =                />
3. <s:file label =                name =                />
4. <s:file label =                name =                />
5. <s:submit />
6. </s:form >

```


如果你希望绑定到数组，**Action** 的代码应类似：

[java] [view plaincopyprint?](#)

```
1. private File[] uploads;
2.   private String[] uploadFileNames;
3.   private String[] uploadContentTypes;
4.
5.   public File[] getUpload() { return this .uploads; }
6.   public void setUpload(File[] upload) { this .uploads = upload; }
7.
8.   public String[] getUploadFileName() { return this .uploadFileNames; }
9.   public void setUploadFileName(String[] uploadFileName) { this .uploadFileName
    s = uploadFileName; }
10.
11.  public String[] getUploadContentType() { return this .uploadContentTypes; }
12.  public void setUploadContentType(String[] uploadContentType) { this .uploadCo
    ntentTypes = uploadContentType; }
```

如果你想绑定到列表，则应类似：

[java] [view plaincopyprint?](#)

```
1. private List < File > uploads = new ArrayList < File > ();
2.   private List < String > uploadFileNames = new ArrayList < String > ();
3.   private List < String > uploadContentTypes = new ArrayList < String > ();
4.
5.   public List < File > getUpload() {
6.       return this .uploads;
7.   }
8.   public void setUpload(List < File > uploads) {
9.       this .uploads = uploads;
10.  }
```

```

11.
12. public List < String > getUploadFileName() {
13.     return this .uploadFileNames;
14. }
15. public void setUploadFileName(List < String > uploadFileNames) {
16.     this .uploadFileNames = uploadFileNames;
17. }
18.
19. public List < String > getUploadContentType() {
20.     return this .uploadContentTypes;
21. }
22. public void setUploadContentType(List < String > contentTypes) {
23.     this .uploadContentTypes = contentTypes;
24. }

```

收集好数据之后，文件上传步骤就和上面单文件的一样了。在这就不重复了。好了，文件上传暂时先说到这里，下一步我们来看一下文件下载。

文件下载

Struts 2 中对文件下载做了直接的支持，相比起自己辛辛苦苦的设置种种 *HTTP* 头来说，现在实现文件下载无疑要简便的多。说起文件下载，最直接的方式恐怕是直接写一个超链接，让地址等于被下载的文件，例如：

` 下载 file1.zip`，之后用户在浏览器里面点击这个链接，就可以进行下载了。但是它有一些缺陷，例如如果地址是一个图片，那么浏览器会直接打开它，而不是显示保存文件的对话框。再比如如果文件名是中文的，它会显示一堆 *URL* 编码过的文件名例如 `%3457...`。而假设你企图这样下载文件：<http://localhost:8080/struts2/download/java> 程序员由笨鸟到菜鸟.doc，*Tomcat* 会告诉你一个文件找不到的 *404* 错误：

HTTP Status 404 - /struts2hello/download/ĬμÍ³ĖμÃ÷.doc。所以在此我们就要用到 **struts** 给我们提供的文件下载了。下面我们就一起来看一下 **struts2** 给我们提供的文件下载：

其实 **struts2** 提供给我们的文件下载已经非常简单化了，编写一个普通的 *Action* 就可以了，只需要提供一个返回 *InputStream* 流的方法，该输入流代表了被下载文件的入口，这个方法用来给被下载的数据提供输入流，意思是从这个流读出来，再写到浏览器那边供下载。这个方法需要由开发人员自己来编写，只需要返回值为 *InputStream* 即可。首先我们来看一下 **jsp** 页面：

[html] view plaincopyprint?

1. `<body>`
2. `<h1>文件下载</h1>`
3. `<!-- 下载链接 -->`
4. `<s:a action= >download</s:a>`
5. `</body>`

页面很简单，就一下下载的连接，然后这个链接链接到我们的 **action** 中，下一步我们来看一下我们的 **action** 的编写：

[java] view plaincopyprint?

1. `public class DownAction extends ActionSupport {`
2. `private static final long serialVersionUID = 1L;`
3. `private String inputPath;`
4. `public String getInputPath() {`
5. `return inputPath;`
6. `}`
7. `public void setInputPath(String inputPath) {`

```

8.      this.inputPath = inputPath;
9.  }
10. /*
11.  * 下载用的 Action 应该返回一个 InputStream 实例，该方法对应在 result 里的
      inputName 属性为
12.  * getDownloadFile
13.  */
14. public InputStream getDownloadFile()
15. {
16.     return ServletActionContext.getServletContext().getResourceAsStream(
17.         "/upload/Struts2.txt");
18. }
19. public String execute() throws Exception
20. {

```

[java] [view plaincopyprint?](#)

```

1. return SUCCESS;

```

[java] [view plaincopyprint?](#)

```

1. }

```

[java] [view plaincopyprint?](#)

```

1. }

```

下面我们在来看一下 `struts.xml` 的配置：

[html] [view plaincopyprint?](#)

```

1. <action name=          class=          >
2.     <!-- 配置结果类型为 stream 的结果 -->
3.     <result name=          type=          >
4.     <!-- 指定被下载文件的文件类型 -->

```

```

5.      <param name=                >text/plain </param>
6.      <!-- 指定下载文件的文件位置 -->
7.      <param name=                >attachment;
      filename=                </param>
8.      <param name=                >downloadFile </param>
9.      </result>
10.    </action>

```

这个 *action* 特殊的地方在于 *result* 的类型是一个流（**stream**），配置 *stream* 类型的结果时，因为无需指定实际的显示的物理资源，所以无需指定 **location** 属性，只需要指定 **inputName** 属性，该属性指向被下载文件的来源，对应着 *Action* 类中的某个属性，类型为 *InputStream*。下面则列出了和下载有关的一些参数列表：

参数说明

contentType

内容类型，和互联网 *MIME* 标准中的规定类型一致，例如 *text/plain* 代表纯文本，*text/xml* 表示 XML，*image/gif* 代表 GIF 图片，*image/jpeg* 代表 JPG 图片

inputName

下载文件的来源流，对应着 *action* 类中某个类型为 *InputStream* 的属性名，例如取值为 *inputStream* 的属性需要编写 *getInputStream()* 方法

contentDisposition

文件下载的处理方式，包括内联(*inline*)和附件(*attachment*)两种方式，而附件方式会弹出文件保存对话框，否则浏览器会尝试直接显示文件。取值为：**attachment;filename="struts2.txt"**，表示文件下载的时候保存的名字应

为 `struts2.txt` 。如果直接写 `filename="struts2.txt"` ，那么默认情况是代表

`inline` ，浏览器会尝试自动打开它，等价于这样的写法：

```
inline; filename="struts2.txt"
```

bufferSize

下载缓冲区的大小

在这里面，`contentType` 属性和 `contentDisposition` 分别对应着 *HTTP* 响应中的头 `Content-Type` 和 `Content-disposition` 头

当然，在很多时候我们一般都不是把文件名写死在 `xml` 配置当中的，我们一般会在 `action` 中定义一个变量 `downfilename`，储存这个文件名，然后再 `xml` 配置：

[html] [view plaincopyprint?](#)

```
1. <param name=                                >attachment;filename=
    </param>
```

我们经常在中文下载中，出现文件名乱码的问题，这个问题是很多人都常见的，具体的很好的解决方法也没有，但我以前用的时候尝试着给他重新编码，这样可以解决大部分时候的中文下载乱码问题，但有时候也不行的。具体重新编码就是：

[java] [view plaincopyprint?](#)

```
1. downFileName = new String(downFileName.getBytes(), "ISO8859-1");
```

好了，`struts2` 文件上传和下载就简单介绍到这，文章中有什么不对或者错误的地方，欢迎大家指正。

分享到：

（七十二）细谈 Spring（四）利用注解实现 spring 基本配置详解

注：由于本人不大习惯注解方式，所以讲解完这里的注解实现基本配置之后，以后就不再单独把注解拿出来讲解了。

五：Spring 注解

1.准备工作

- (1)导入 common-annotations.jar
- (2)导入 schema 文件 文件名为 spring-context-2.5.xsd
- (3)在 xml 的 beans 节点中配置

2.xml 配置工作

[\[html\] view plaincopyprint?](#)

1. `<?xml version= encoding= ?>`
2. `<beans xmlns=`
3. `xmlns:xsi=`
4. `xmlns:context=`
5. `xsi:schemaLocation="http://www.springframework.org/schema/beans`
6. `http://www.springframework.org/schema/beans/spring-beans-2.5.xsd`
7. `http://www.springframework.org/schema/context`
8. `http://www.springframework.org/schema/context/spring-context-2.5.xsd"`
9. `default-lazy-init= >`
10. `<!--将针对注解的处理器配置好 -->`
11. `<context:annotation-config />`
12. `<!-- 使用 annotation 定义事务-->`

13. `<tx:annotation-driven transaction-manager="tx:annotation-driven transaction-manager" proxy-target-class=""/>`
14. `<!--使用 annotation 自动注册 bean,并检查@Required,@Autowired 的属性已被注入 base-package 为需要扫描的包（含所有子包）-->`
15. `<context:component-scan base-package="context:component-scan base-package" = />`
16.
17. `<beans>`

注： `<context:component-scan base-package="*. *" />`

该配置隐式注册了多个对注解进行解析的处理器，如：

AutowiredAnnotationBeanPostProcessor
 CommonAnnotationBeanPostProcessor
 PersistenceAnnotationBeanPostProcessor
 RequiredAnnotationBeanPostProcessor

其实，注解本身做不了任何事情，和 XML 一样，只起到配置的作用，主要在于背后强大的处理器，其中就包括了 `<context:annotation-config/>` 配置项里面的注解所使用的处理器，所以配置了

`<context:component-scan base-package="">` 之后，便无需再配置
`<context:annotation-config>`

1. 在 java 代码中使用 `@Autowired` 或 `@Resource` 注解方式进行装配，这两个注解的区别是：`@Autowired` 默认按类型装配，`@Resource` 默认按名称装配，当找不到名称匹配的 bean 才会按类型装配。

`@Autowired` 一般装配在 set 方法之上，也可以装配在属性上边，但是在属性上边配置，破坏了 java 的封装，所以一般不建议使用

@Autowired 是根据类型进行自动装配的。如果当 Spring 上下文中存在不止一个所要装配类型的 bean 时，就会抛出 **BeanCreationException** 异常；如果 Spring 上下文中不存在所要装配类型的 bean，也会抛出 **BeanCreationException** 异常。我们可以使用 **@Qualifier** 配合 **@Autowired** 来解决这些问题。

[java] [view plaincopyprint?](#)

```
1. @Autowired
2. public void setUserDao(@Qualifier("userDao") UserDao userDao) {
3.     this.userDao = userDao;
4. }
```

这样，Spring 会找到 id 为 userDao 的 bean 进行装配。

可能不存在 UserDao 实例

[java] [view plaincopyprint?](#)

```
1. @Autowired(required = false)
2. public void setUserDao(UserDao userDao) {
3.     this.userDao = userDao;
4. }
```

2.@Resource（JSR-250 标准注解，推荐使用它来代替 Spring 专有的

@Autowired 注解）Spring 不但支持自己定义的 **@Autowired** 注解，还支持几个由 JSR-250 规范定义的注解，它们分别是 **@Resource**、**@PostConstruct** 以及 **@PreDestroy**。

@Resource 的作用相当于 **@Autowired**，只不过 **@Autowired** 按 **byType** 自动注入，而 **@Resource** 默认按 **byName** 自动注入罢了。**@Resource** 有两个属性

是比较重要的，分别是 `name` 和 `type`，Spring 将 `@Resource` 注解的 `name` 属性解析为 `bean` 的名字，而 `type` 属性则解析为 `bean` 的类型。所以如果使用 `name` 属性，则使用 `byName` 的自动注入策略，而使用 `type` 属性时则使用 `byType` 自动注入策略。如果既不指定 `name` 也不指定 `type` 属性，这时将通过反射机制使用 `byName` 自动注入策略

`@Resource` 装配顺序

- 1 如果同时指定了 `name` 和 `type`，则从 `Spring` 上下文中找到唯一匹配的 `bean` 进行装配，找不到则抛出异常
- 2 如果指定了 `name`，则从上下文中查找名称 (`id`) 匹配的 `bean` 进行装配，找不到则抛出异常
- 3 如果指定了 `type`，则从上下文中找到类型匹配的唯一 `bean` 进行装配，找不到或者找到多个，都会抛出异常
- 4 如果既没有指定 `name`，又没有指定 `type`，则自动按照 `byName` 方式进行装配（见 2）；如果没有匹配，则回退为一个原始类型（`UserDao`）进行匹配，如果匹配则自动装配；

3. `@PostConstruct` (JSR-250)

在方法上加上注解 `@PostConstruct`，这个方法就会在 `Bean` 初始化之后被 `Spring` 容器执行（注：`Bean` 初始化包括，实例化 `Bean`，并装配 `Bean` 的属性（依赖注入））。

它的一个典型的应用场景是，当你需要往 `Bean` 里注入一个其父类中定义的属性，而你又无法复写父类的属性或属性的 `setter` 方法时，如：

[Java] [view plaincopyprint?](#)

```

1. public class UserDaoImpl extends HibernateDaoSupport implements UserDao {

2.     private SessionFactory mySessionFactory;

3.     @Resource

4.     public void setMySessionFactory(SessionFactory sessionFactory) {

5.         this.mySessionFactory = sessionFactory;

6.     }

7.     @PostConstruct

8.     public void injectSessionFactory() {

9.         super.setSessionFactory(mySessionFactory);

10.    } }

```

这里通过@PostConstruct, 为 UserDaoImpl 的父类里定义的一个 sessionFactory 私有属性, 注入了我们自己定义的 sessionFactory (父类的 setSessionFactory 方法为 final, 不可复写), 之后我们就可以通过调用 super.getSessionFactory() 来访问该属性了。

4.@PreDestroy (JSR-250)

在方法上加上注解@PreDestroy, 这个方法就会在 Bean 初始化之后被 Spring 容器执行。由于我们当前还没有需要用到它的场景, 这里不去演示。其用法同@PostConstruct。

5.使用 Spring 注解完成 Bean 的定义

以上我们介绍了通过@Autowired 或@Resource 来实现在 Bean 中自动注入的功能, 下面我们将介绍如何注解 Bean, 从而从 XML 配置文件中完全移除 Bean 定义的配置。

@Component: 只需要在对应的类上加上一个@Component 注解，就将该类定义为一个 Bean 了：

[java] [view plaincopyprint?](#)

```
1. @Component
2. public class UserDaoImpl extends HibernateDaoSupport implements UserDao {
3.     ...
4. }
```

使用@Component 注解定义的 Bean，默认的名称（id）是小写开头的非限定类名。如这里定义的 Bean 名称就是 userDaoImpl。你也可以指定 Bean 的名称：

@Component("userDao")

@Component 是所有受 Spring 管理组件的通用形式，Spring 还提供了更加细化的注解形式：**@Repository**、**@Service**、**@Controller**，它们分别对应存储层 Bean，业务层 Bean，和展示层 Bean。目前版本（2.5）中，这些注解与 **@Component** 的语义是一样的，完全通用，在 Spring 以后的版本中可能会给它们追加更多的语义。所以，我们推荐使用 **@Repository**、**@Service**、**@Controller** 来替代 **@Component**。

6.使用<context:component-scan />让 Bean 定义注解工作起来

[html] [view plaincopyprint?](#)

```
1. <PRE class=      name=      ><beans xmlns=
2.      xmlns:xsi=
3.      xmlns:context=
4.      xsi:schemaLocation="http://www.springframework.org/schema/beans
```

5. <http://www.springframework.org/schema/beans/spring-beans-2.5.xsd>
6. <http://www.springframework.org/schema/context>
7. <http://www.springframework.org/schema/context/spring-context-2.5.xsd>
8. `<context:component-scan base-package= />`
9. `</beans> </PRE>
`
10. `
`
11. `<PRE></PRE>`
12. `
`
13. `
`
14. `<P></P>`
15. `<P style= >`
 这里，所有通过
`<bean><SPAN styl`
`e= >元素定义`
`Bean<SP`
`AN style= >的配置内容已经被移除，仅需要添加一行`
`<context:compon`
`ent-scan />配置就解决所有问题`
 了
`——Spring XML</`
`SPAN>配置文件得到了极致的简化（当然配`
 置元数据还是需要的，只不过以注释形式存在罢了）。
`<context:compon`
`ent-scan />的`
`base-package</SP`
`AN>属性指定了需要扫描的类包，类包及其`
 递归子包中所有的类都会被处理。 `
`
16. `<context:component-scan />还允许定义过`
 滤器将基包下的某些类纳入或排除。
`Spring<S`
`PAN style= >支持以下`
`4<SPAN`
`style= >种类型的过滤方式： </P>`
17. `<P style= >`
 · 过滤器类型 表达式范例 说明 `</P>`

18. `<P style=` `>`
 · 注
 解 `org.example.SomeAnnotation </SP`
`AN>将所有使用`
`SomeAnnotation<S`
`PAN style=` `>注解的类过滤出来</P>`
19. `<P style=` `>`
 · 类名指
 定 `org.example.SomeClass <`
`SPAN style=` `>过滤指定的类</P>`
20. `<P style=` `>`
 · 正则表达
 式 `com\.\kedacom\.\spring\.\annotation\.`
`web\.\.* 通过正则表达式过滤一些`
`类</P>`
21. `<P style=` `>`
 · AspectJ`表达`
 式 `org.example.*Service+ </`
`SPAN>通过`
`AspectJ<SPAN styl`
`e=` `>表达式过滤一些类</P>`
22. `<P>7.</`
`SPAN>使用`
`@Scope<SPAN st`
`yle=` `>来定义`
`Bean<SP`
`AN style=` `>的作用范围
`
23. 在使用
`XML<SPAN style=`
`>定义`
`Bean<SP`
`AN style=` `>时，我们可能还需要通过`
`bean<SP`
`AN style=` `>的`
`scope<S`

```

    <SPAN style=
    </SPAN><SPAN style=
    </SPAN><SPAN style=
    <SPAN style=
    </SPAN></SPAN></P>
24. <P style=
    </SPAN></P>
25. <PRE class=
    name=
    >@Scope("session")
26. @Component()
27. public class UserSessionBean implements Serializable {
28.     ...
29. } </PRE>
30. <P></P>
31. <P style=
    <BR>
32. <BR>
33. <BR>
34. <BR>
35. <BR>
36. <BR>
37. </SPAN></P>
38. <P></P>

```

>属性来定义一个

>Bean<SP

>的作用范围，我们同样可以通过

>@Scope<

>注解来完成这项工作：

（七十三）细谈 Spring（五）spring 之 AOP 底层大揭秘

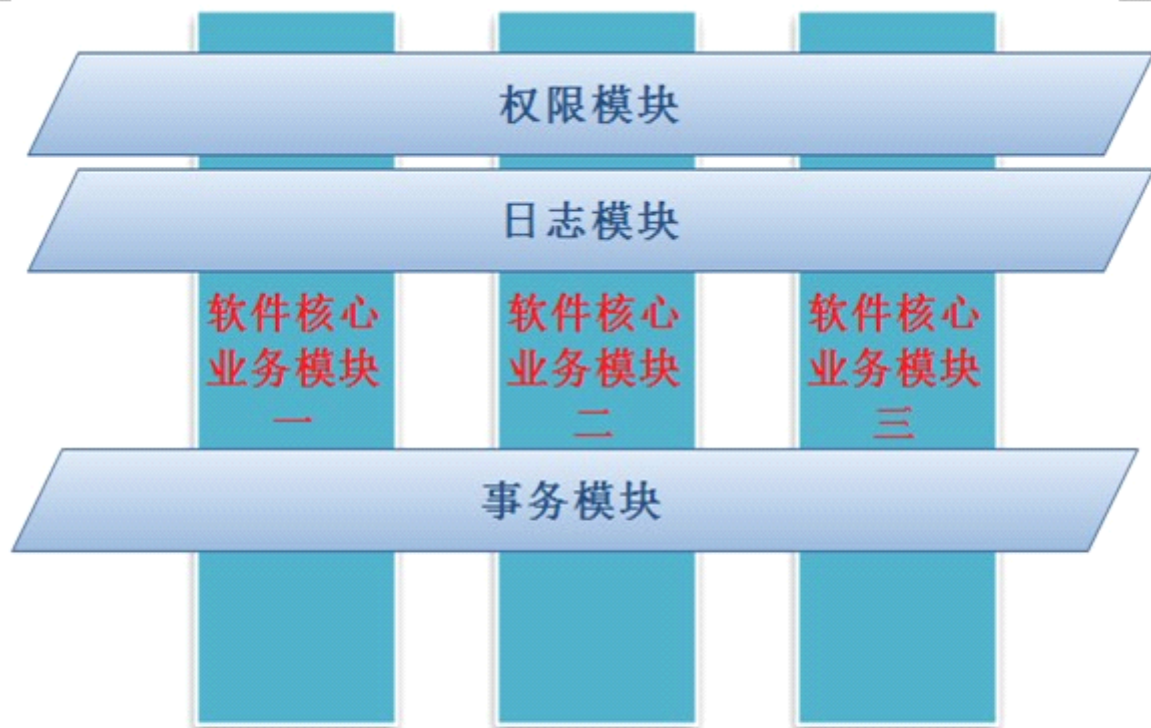
众所周知，java 是面向对象语言的有力代表，提到 java 我们就会立即想到面向对象，提到面向对象我们就会想到 java。然而面向对象也并非完美无缺的，它更侧重于对象层次结构方面的东西，对于如何更好的管理对象行为内部结构，还存在着些许不足。那么我们如何使这个问题的得到更完美的解决呢？

答案就是 AOP。

AOP: Aspect-Oriented Programming. AOP 是 OOP 的补充，是 GOF 的延续。我们知道设计模式是对于面向对象设计中经验的总结，它孜孜不断追求的就是调用者与被调用者之间的解耦。有了设计模式我们可以更有效的利用面向对象的特性，使得整个软件设计更加灵活、优雅。但是设计模式是基于面向对象的思想而形成的，更多的时候关注的是对象层次的东西，在解决对象行为内部问题方面却有些不足。AOP 的出现恰恰就是对面向对象思想做出了完美的补充。

说到 **AOP**，我们就不得不来提一下软件的纵向和横向问题。从纵向结构来看就是我们软件系统的各个模块，它主要负责处理我们的核心业务（例如商品订购、购物车查看）；而从横向结构来看，我们几乎每个系统又包含一些公共模块（例如权限、日志模块等）。这些公共模块分布于我们各个核心业务之中（例如订购和查看商品明细的过程都需要检查用户权限、记录系统日志等）。这样一来不仅在开发过程中要处处关注公共模块的处理而且开发

后维护起来也是十分麻烦。而有了 AOP 之后将应用程序中的商业逻辑同对其提供支持的通用服务进行分离，使得开发人员可以更多的关注核心业务开发。



软件纵向与横向结构

下面我们就以一个简单的例子来看一下 AOP 吧！比如说,我们现在要开发的一个应用里面有很多的业务方法,但是,我们现在要对这个方法的执行做全面监控,或部分监控.也许我们就会在要一些方法前去加上一条日志记录,我们写个例子看看我们最简单的解决方案

我们先写一个接口 IHello.java 代码如下：

[java] [view plaincopyprint?](#)

```
1. public interface IHello{  
2.    /** *//**
```

3. * 假设这是一个业务方法
4. *@param name
5. */
6. void sayHello(String name);
7. }

里面有个方法,用于输入"Hello" 加传进来的姓名;我们去写个类实现 IHello 接

口

[java] [view plaincopyprint?](#)

1. public class Helloimplements IHello{
- 2.
3. public void sayHello(String name){
4. System.out.println("Hello" + name);
5. }
6. }

现在我们要为这个业务方法加上日志记录的业务,我们在不改变原代码的情况下,我们会去怎么做呢?
也许,你会去写一个类去实现 IHello 接口,并依赖 Hello 这个类.代码如下:

[java] [view plaincopyprint?](#)

1. public class HelloProxyimplements IHello{
2. private IHello hello;
- 3.
4. public HelloProxy(IHello hello){
5. this.hello= hello;
6. }
- 7.
8. public void sayHello(String name){
9. Logger.logging(Level.DEBUG,"sayHello method start.");
10. hello.sayHello(name);

```

11. Logger.logging(Level.INFO,"sayHello method end!");
12.
13. }
14.}

```

从上面的代码我们可以看出,hello 对象是被 HelloProxy 这个所谓的代理态所创建的.这样,如果我们以后要把日志记录的功能去掉.那我们只要把得到 hello 对象的的具体实现改为 Hello 的就可以。上面的代码 就是对 AOP 的最简单的视线，但是我们接下来想，如果我们要在很多业务逻辑之前加日志的话，那么,我们是不是要去写很多个 HelloProxy 这样的类呢.没错,是的.其实也是一种很麻烦的事.在 jdk1.3 以后,jdk 跟我们提供了一个 API java.lang.reflect.InvocationHandler 的类. 这个类可以让我们在 JVM 调用某个类的方法时动态的为些方法做些什么事.让我们把以上的代码改一下来看效果.

同样,我们写一个 IHello 的接口和一个 Hello 的实现类.在接口中.我们定义两个方法;代码如下：

IHello.java

[java] [view plain](#)[copy](#)[print?](#)

```

1. package sinosoft.dj.aop.proxyaop;
2.
3. public interface IHello{
4.     /** **/**
5.     * 业务处理 A 方法
6.     * @param name

```

```

7. */
8. void sayHello(String name);
9. /** */
10. * 业务处理 B 方法
11. * @param name
12. */
13. void sayGoogBye(String name);
14. }

```

Hello.java

[java] [view plaincopyprint?](#)

```

1. package sinosoft.dj.aop.proxyaop;
2.
3. public class Helloimplements IHello{
4.
5.     public void sayHello(String name){
6.         System.out.println("Hello" + name);
7.     }
8.     public void sayGoogBye(String name){
9.         System.out.println(name+" GoodBye!");
10.    }
11.}

```

我们一样的去写一个代理类.只不过.让这个类去实现

java.lang.reflect.InvocationHandler 接口,代码如下:

[java] [view plaincopyprint?](#)

```

1. package sinosoft.dj.aop.proxyaop;

```

```

2.
3. import java.lang.reflect.InvocationHandler;
4. import java.lang.reflect.Method;
5. import java.lang.reflect.Proxy;
6.
7. public class DynaProxyHelloimplements InvocationHandler{
8.
9.     /** **/**
10.    * 要处理的对象(也就是我们要在方法的前后加上业务逻辑的对象,如例子中的
        Hello)
11.    */
12.    private Object delegate;
13.
14.    /** **/**
15.    * 动态生成方法被处理过后的对象 (写法固定)
16.    *
17.    * @param delegate
18.    * @param proxy
19.    * @return
20.    */
21.    public Object bind(Object delegate){
22.        this.delegate= delegate;
23.        return Proxy.newProxyInstance(
24.            this.delegate.getClass().getClassLoader(),this.delegate
25.            .getClass().getInterfaces(),this);
26.    }
27.    /** **/**
28.    * 要处理的对象中的每个方法会被此方法送去 JVM 调用,也就是说,要处理的对象的
        方法只能通过此方法调用
29.    * 此方法是动态的,不是手动调用的
30.    */
31.    public Object invoke(Object proxy, Method method, Object[] args)
32.        throws Throwable{
33.        Object result= null;
34.        try {

```

```
35.//执行原来的方法之前记录日志
36. Logger.logging(Level.DEBUG, method.getName()+ " Method end .");
37.
38.//JVM 通过这条语句执行原来的方法(反射机制)
39. result= method.invoke(this.delegate, args);
40.//执行原来的方法之后记录日志
41. Logger.logging(Level.INFO, method.getName()+ " Method Start!");
42. } catch (Exception e){
43. e.printStackTrace();
44. }
45.//返回方法返回值给调用者
46. return result;
47.}}
```

从上面的例子我们看出.只要你是采用面向接口编程,那么,你的任何对象的方法执行之前要加上记录日志的操作都是可以的.他(DynaPoxyHello)自动去代理执行被代理对象(Hello)中的每一个方法,一个 `java.lang.reflect.InvocationHandler` 接口就把我们的代理对象和被代理对象解藕了

（七十四）细谈 Spring（六）spring 之 AOP 基本概念和配置详解

首先我们来看一下官方文档给我们的关于 AOP 的一些概念性词语的解释：

切面（Aspect）：一个关注点的模块化，这个关注点可能会横切多个对象。

事务管理是 J2EE 应用中一个关于横切关注点的很好的例子。在 Spring AOP 中，切面可以使用[基于模式](#)或者基于 **Aspect** 注解方式来实现。通俗点说就是我们加入的切面类（比如 log 类），可以这么理解。

连接点（Joinpoint）：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。在 Spring AOP 中，一个连接点总是表示一个方法的执行。[通俗的说就是加入切点的那个点](#)

通知（Advice）：在切面的某个特定的连接点上执行的动作。其中包括了“around”、“before”和“after”等不同类型的通知（通知的类型将在后面部分进行讨论）。许多 AOP 框架（包括 Spring）都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。

切入点（Pointcut）：匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是 AOP 的核心：Spring 缺省使用 AspectJ 切入点语法。

引入（Introduction）：用来给一个类型声明额外的方法或属性（也被称为连接类型声明（inter-type declaration））。Spring 允许引入新的接口（以及一个

对应的实现) 到任何被代理的对象。例如, 你可以使用引入来使一个 bean 实现 IsModified 接口, 以便简化缓存机制。

目标对象 (Target Object): 被一个或者多个切面所通知的对象。也被称做被通知 (advised) 对象。既然 Spring AOP 是通过运行时代理实现的, 这个对象永远是一个被代理 (proxied) 对象。

AOP 代理 (AOP Proxy): AOP 框架创建的对象, 用来实现切面契约 (例如通知方法执行等等)。在 Spring 中, AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。

织入 (Weaving): 把切面连接到其它的应用程序类型或者对象上, 并创建一个被通知的对象。这些可以在编译时 (例如使用 AspectJ 编译器), 类加载时和运行时完成。Spring 和其他纯 Java AOP 框架一样, 在运行时完成织入。

通知类型:

前置通知 (Before advice): 在某连接点之前执行的通知, 但这个通知不能阻止连接点之前的执行流程 (除非它抛出一个异常)。

后置通知 (After returning advice): 在某连接点正常完成后执行的通知: 例如, 一个方法没有抛出任何异常, 正常返回。

异常通知 (After throwing advice): 在方法抛出异常退出时执行的通知。

最终通知 (After (finally) advice): 当某连接点退出的时候执行的通知 (不论是正常返回还是异常退出)。

环绕通知 (Around Advice): 包围一个连接点的通知, 如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它

也会选择是否继续执行连接点或直接返回它自己的返回值或抛出异常来结束执行。

环绕通知是最常用的通知类型。和 AspectJ 一样，Spring 提供所有类型的通知，我们推荐你使用尽可能简单的通知类型来实现需要的功能。例如，如果你只是需要一个方法的返回值来更新缓存，最好使用后置通知而不是环绕通知，尽管环绕通知也能完成同样的事情。用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。比如，你不需要在 JoinPoint 上调用用于环绕通知的 proceed() 方法，就不会有调用的问题。

spring AOP 的实现

在 spring2.5 中，常用的 AOP 实现方式有两种。第一种是基于 xml 配置文件方式的实现，第二种是基于注解方式的实现。接下来，以具体的示例来讲解这两种方式的使用。下面我们要用到的实例是一个注册，就有用户名和密码，我们利用 AOP 来实现在用户注册的时候实现在保存数据之前和之后或者是抛出异常时，在这些情况下都给他加上日志。在这里我们只讲解 AOP，所以我只把关键代码贴出来，不相干的就不贴了。

首先我们来看一下业务逻辑 *service* 层：

[java] [view plain](#)[copy](#)[print?](#)

```
1. /**
2.  * RegisterService 的实现类
3.  * @author 曹胜欢 */
4. public class RegisterServiceImpl implements RegisterService {
5.     private RegisterDao registerDao;
6.     public RegisterServiceImpl() {}
```

```

7.  /** 带参数的构造方法 */
8.  public RegisterServiceImpl(RegisterDao registerDao){
9.      this.registerDao =registerDao;
10. }
11. public void save(String loginname, String password) {
12.     registerDao.save(loginname, password);
13.     throw new RuntimeException("故意抛出一个异常。。。");
14. }
15. /** set 方法 */
16. public void setRegisterDao(RegisterDao registerDao) {
17.     this.registerDao = registerDao;
18.}}

```

对于业务系统来说，**RegisterServiceImpl** 类就是目标实现类，它的业务方法，如 **save()** 方法的前后或代码会出现异常的地方都是 **AOP** 的连接点。

下面是日志服务类的代码：

[java] [view plaincopyprint?](#)

```

1. /**
2.  * 日志切面类
3.  * @author 曹胜欢
4.  */
5. public class LogAspect {
6.     //任何通知方法都可以将第一个参数定义为 org.aspectj.lang.JoinPoint 类型
7.     public void before(JoinPoint call) {
8.         //获取目标对象对应的类名
9.         String className = call.getTarget().getClass().getName();
10.        //获取目标对象上正在执行的方法名
11.        String methodName = call.getSignature().getName();

```

```

12.     System.out.println("前置通知:" + className + "类的" + methodName + "方法
        开始了");
13. }
14. public void afterReturn() {
15.     System.out.println("后置通知:方法正常结束了");
16. }
17. public void after(){
18.     System.out.println("最终通知:不管方法有没有正常执行完成,一定会返回的");
19. }
20. public void afterThrowing() {
21.     System.out.println("异常抛出后通知:方法执行时出异常了");
22. }
23. //用来做环绕通知的方法可以第一个参数定义为
        org.aspectj.lang.ProceedingJoinPoint 类型
24. public Object doAround(ProceedingJoinPoint call) throws Throwable {
25.     Object result = null;
26.     this.before(call);//相当于前置通知
27.     try {
28.         result = call.proceed();
29.         this.afterReturn(); //相当于后置通知
30.     } catch (Throwable e) {
31.         this.afterThrowing(); //相当于异常抛出后通知
32.         throw e;
33.     }finally{
34.         this.after(); //相当于最终通知
35.     }
36.     return result;
37. }
38.}

```

这个类属于业务服务类，如果用 **AOP** 的术语来说，它就是一个切面类，它定义了许多通知。**Before()**、**afterReturn()**、**after()**和 **afterThrowing()**这些方法都是通知。

下面我们就来看具体配置，首先来看一下：

<1>.基于 xml 配置文件的 AOP 实现：这种方式在实现 AOP 时，有 4 个步骤。

[html] view plaincopyprint?

```
1. <?xml version=      encoding=      ?>
2. <beans xmlns=
3.     xmlns:xsi=
4.     xmlns:aop=
5.     xsi:schemaLocation="
6.         http://www.springframework.org/schema/beans http://www.springframewor
           k.org/schema/beans/spring-beans-2.5.xsd
7.         http://www.springframework.org/schema/aop http://www.springframework.
           org/schema/aop/spring-aop-2.5.xsd>
8. <bean id=            class=            />
9. <bean id=            class=            >
10. <property name=            ref=            />
11. </bean>
12. <!-- 日志切面类 -->
13. <bean id=            class=            />
14. <!-- 第 1 步： AOP 的配置 -->
15. <aop:config>
16.     <!-- 第 2 步： 配置一个切面 -->
17.     <aop:aspect id=            ref=            >
18.         <!-- 第 3 步： 定义切入点,指定切入点表达式 -->
19.         <aop:pointcut id=
20.             expression=            />
21.         <!-- 第 4 步： 应用前置通知 -->
22.         <aop:before method=            pointcut-ref=            />
23.         <!-- 第 4 步： 应用后置通知 -->
24.         <aop:after-returning method=            pointcut-ref=            />
25.         <!-- 第 4 步： 应用最终通知 -->
26.         <aop:after method=            pointcut-ref=            />
```

```

27.      <!-- 第 4 步：应用抛出异常后通知 -->
28.      <aop:after-throwing method=          pointcut-ref=          />
29.      <!-- 第 4 步：应用环绕通知 -->
30.      <!--
31.      <aop:around method="doAround" pointcut-ref="allMethod" />
32.      -->
33.  </aop:aspect>
34. </aop:config>
35.</beans>

```

上述配置针对切入点应用了前置、后置、最终，以及抛出异常后通知。这样在测试执行 `RegisterServiceImpl` 类的 `save()` 方法时，控制台会有如下结果输出：

前置通知：`com.zxf.service.RegisterServiceImpl` 类的 `save` 方法开始了。

针对 `MySQL` 的 `RegisterDao` 实现中的 `save()` 方法。

后置通知:方法正常结束了。

最终通知:不管方法有没有正常执行完成，一定会返回的。

下面我们在来看一下第二种配置方式：

<2>基于注解的 AOP 的实现

首先创建一个用来作为切面的类 `LogAnnotationAspect`，同时把这个类配置在 `spring` 的配置文件中。

在 `spring2.0` 以后引入了 `JDK5.0` 的注解 `Annotation` 的支持，提供了对 `AspectJ` 基于注解的切面的支持，从而 更进一步地简化 AOP 的配置。具体的步骤有两步。

Spring 的配置文件是如下的配置：

[html] view plaincopyprint?

```
1. <?xml version=      encoding=      ?>
2. <beans xmlns=
3.      xmlns:xsi=
4.      xmlns:aop=
5.      xsi:schemaLocation="
6.          http://www.springframework.org/schema/beans http://www.springframewor
k.org/schema/beans/spring-beans-2.5.xsd
7.          http://www.springframework.org/schema/aop http://www.springframework.
org/schema/aop/spring-aop-2.5.xsd>
8. <bean id=      class=      />
9. <bean id=      class=      >
10. <property name=      ref=      />
11. </bean>
12. <!-- 把切面类交由 Spring 容器来管理 -->
13. <bean id=      class=      />
14. <!-- 启用 spring 对 AspectJ 注解的支持 -->
15. <aop:aspectj-autoproxy/>
16. </beans>
```

这是那个切面的类 **LogAnnotationAspect**

[java] view plaincopyprint?

```
1. /**
2.  * 日志切面类
3.  */
4. @Aspect //定义切面类
5. public class LogAnnotationAspect {
```

```

6.  @SuppressWarnings("unused")
7.  //定义切入点，提供一个方法，这个方法的名字就是改切入点的 id
8.  @Pointcut("execution(* com.zxf.service.*(..))")
9.  private void allMethod(){}
10. //针对指定的切入点表达式选择的切入点应用前置通知
11. @Before("execution(* com. zxf.service.*(..))")
12. public void before(JoinPoint call) {
13.     String className = call.getTarget().getClass().getName();
14.     String methodName = call.getSignature().getName();
15.     System.out.println("【注解-前置通知】:" + className + "类的"
16.         + methodName + "方法开始了");
17. }
18. //访问命名切入点来应用后置通知
19. @AfterReturning("allMethod()")
20. public void afterReturn() {
21.     System.out.println("【注解-后置通知】:方法正常结束了");
22. }
23. //应用最终通知
24. @After("allMethod()")
25. public void after(){
26.     System.out.println("【注解-最终通知】:不管方法有没有正常执行完成,"
27.         + "一定会返回的");
28. }
29. //应用异常抛出后通知
30. @AfterThrowing("allMethod()")
31. public void afterThrowing() {
32.     System.out.println("【注解-异常抛出后通知】:方法执行时出异常了");
33. }
34. //应用周围通知
35. //@Around("allMethod()")
36. public Object doAround(ProceedingJoinPoint call) throws Throwable{
37.     Object result = null;
38.     this.before(call);//相当于前置通知
39.     try {
40.         result = call.proceed();

```

```
41.         this.afterReturn(); //相当于后置通知
42.     } catch (Throwable e) {
43.         this.afterThrowing(); //相当于异常抛出后通知
44.         throw e;
45.     }finally{
46.         this.after(); //相当于最终通知
47.     }
48.     return result;
49. }
50.}
```

备注：输出结果和前面的一样。

分享到：

（七十五）细谈 struts2（十四）struts2+ajax 实现异步验证

由于老师布置作业的需要，在添加管理员的时候，要实现验证添加的管理员的用户名是否在数据库中已经存在，然后再客户端给用户一个提示。我首先想到的就是利用 **ajax** 实现异步验证技术，由于利用的 **ssh** 框架，所以在这要对 **struts2** 和 **ajax** 进行整合，由于我还没把 **ajax** 的一些知识总结出来，所以在这也不提了，有关 **ajax** 的详细内容将会在以后的博客中写出来。现在我们就以我做的这个添加管理员，验证管理员的用户名是否存在来说一下这个 **struts2+ajax** 实现异步验证技术。

首先我们来看一下我们的 **form** 表单：

[html] [view plaincopyprint?](#)

```
1. <td>
2. 用户名
3. </td>
4. <td>
5. <input type="text" name="username" value=""
6. onblur="checkUsername()" />
7. </td>
8. <td>
9. <span id="usernameError" style="color: red;"></span>
10.</td>
11.</tr>
```

```

12.<tr bgColor=          >
13.<td>
14.密码
15.</td>
16.<td>
17.<input type=          name=          value=  />
18.</td>
19.<td>
20.<span></span>
21.</td>
22.</tr>

```

我们可以看到当我们的用户名的文本域注册了一个 `onblur` 事件，当用户名这个文本域失去焦点的时候我们就会让他去执行 `checkusername` 方法，好，下面让我们来看一下我们的 `js` 是怎么实现的 `ajax`：

[javascript] [view plaincopyprint?](#)

```

1. <script type="text/javascript">
2. var xmlhttpRequest = null; //声明一个空对象以接收 XMLHttpRequest 对象
3. function checkusername(field, url) {
4. var uername = field.value;
5. if (uername == "" || uername.length < 3) {
6. document.getElementById("namemessage").innerHTML = "用户名应该不小于 3 位";
7. return;
8. } else {
9. if (window.ActiveXObject) // IE 浏览器
10.{
11.xmlHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
12.} else if (window.XMLHttpRequest) //除 IE 外的其他浏览器实现
13.{
14.xmlHttpRequest = new XMLHttpRequest();

```

```

15.}
16.if (null != xmlHttpRequest) {
17.//当利用 get 方法访问服务器端时带参数的话，直接在"AjaxServlet"后面加参
    数，          下面 send 方法为参数 null 就可以，用 post 方法这必须在把参数加
    在 send 参数内，如下
18.xmlHttpRequest.open("get", url+"?admin.username="+uername, true);
19.//关联好 ajax 的回调函数
20.xmlHttpRequest.onreadystatechange = ajaxCallback;
21.//真正向服务器端发送数据
22.// 使用 post 方式提交，必须要加上如下一行,get 方法就不必加此句
23.xmlHttpRequest.setRequestHeader("Content-Type",
24."application/x-www-form-urlencoded");
25.xmlHttpRequest.send(null);
26.}
27.}
28.}
29.function ajaxCallback() { //ajax 一次请求会改变四次状态，所以我们在第四次(即一次
    请求结束)进行处理就 OK,
30.if (xmlHttpRequest.readyState == 4) { //请求成功
31.if (xmlHttpRequest.status == 200) {
32.var responseText = xmlHttpRequest.responseText;
33.document.getElementById("namemessage").innerHTML = responseText;
34.}
35.}
36.}
37.</script>

```

通过上面的注释我想大家应该能看懂一些内容吧，我们首先去验证填写的内容是否为空，如果为空就给用户以提示。如果不为空的话就去判断一下当前的浏览器，然后根据浏览器去设置 xmlHttpRequest 对象，

xmlHttpRequest 对象是什么呢？XMLHttpRequest 对象用于在后台与服务器交换数据的对象，他主要的作用：

- 在不重新加载页面的情况下更新网页
- 在页面已加载后从服务器请求数据
- 在页面已加载后从服务器接收数据
- 在后台向服务器发送数据

XMLHttpRequest 是 Ajax 最核心的对象，它有以下几个重要的方法或属性：

- open(): 建立到服务器的新请求。
- send(): 向服务器发送请求。
- abort(): 退出当前请求。
- readyState: 提供当前 HTML 的就绪状态。
- responseText: 服务器返回的请求响应文本。

其中 XMLHttpRequest 对象的 open() 方法有以下五个参数：

- request-type: 发送请求的类型。典型的值是 GET 或 POST，但也可以发送 HEAD 请求。
- url: 要连接的 URL。
- asynch: 如果希望使用异步连接则为 true，否则为 false。该参数是可选的，默认为 true。
- username: 如果需要身份验证，则可以在此指定用户名。该可选参数没有默认值。

●**password**: 如果需要身份验证, 则可以在此指定口令。该可选参数没有默认值。

通常使用其中的前三个参数。事实上, 即使需要异步连接, 也应该指定第三个参数为“**true**”。这是默认值, 但坚持明确指定请求是异步的还是同步的更容易理解。

得到 **XMLHttpRequest** 对象之后, 我们就利用这个对象去后台执行我们的请求, 在执行我们请求的时候一定要注意关联好我们的回调函数:

xmlHttpRequest.onreadystatechange = ajaxCallback;这里的回调函数的名字可以随便起, 并不是固定死的。我们可以看到我们上面的程序请求是发送给了 **AdminAction** 中的 **exists** 方法了, 好, 下面我们去 **action** 方法里面去看一下:

[java] [view plaincopyprint?](#)

```
1. public String exists() throws Exception{
2.     System.out.println(admin==null);
3.     boolean boo=dao.exists(admin.getUsername());
4.     //获取原始的 PrintWriter 对象,以便输出响应结果,而不用跳转到某个试图
5.     HttpServletResponse response = ServletActionContext.getResponse();
6.     //设置字符集
7.     response.setCharacterEncoding("UTF-8");
8.     PrintWriter out = response.getWriter();
9.     if(boo){
10.        //直接输入响应的内容
11.        out.println("**用户名已存在**");
12.        /**格式化输出时间**/
13.        out.flush();
14.        out.close();
15.}
```

```
16.out.println("**用户名可用*");
17.return null;
18.}
```

熟悉 **ajax** 的同学看到这段代码应该很清楚了吧。这里主要是利用了 **PrintWriter** 来把我们的后台信息输出到我们的前台，这里我 就不详细解释了。好了，写到这，我们这个利用 **struts2+ajax** 实现的我们的异步验证。下面就是具体的实现效果：

添加管理员

用户名	<input type="text" value="123"/>	
密码	<input type="text"/>	
姓名	<input type="text"/>	
<input type="button" value="提交"/>		<input type="button" value="重置"/>

（七十六）细谈 Hibernate（十八）悲观锁和乐观锁解决 hibernate 并发

锁（ locking ），这个概念在我们学习多线程的时候曾经接触过，其实这里的锁和多线程里面处理并发的锁是一个道理，都是暴力的把资源归为自己所有。这里我们用到锁的目的就是通过一些机制来保证一些数据在某个操作过程中不会被外界修改，这样的机制，在这里，也就是所谓的“锁”，即给我们选定的目标数据上锁，使其无法被其他程序修改。**Hibernate** 支持两种锁机制：即通常所说的“悲观锁（**Pessimistic Locking** ）”和“乐观锁（**Optimistic Locking** ）”。

悲观锁（ Pessimistic Locking ）

悲观锁，正如其名，他是对数据库而言的，数据库悲观了，他感觉每一个对他操作的程序都有可能产生并发。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改

数据)。

一个典型的倚赖数据库的悲观锁调用：

[sql] [view plaincopyprint?](#)

```
1. select * from account where name="Erica" for update
```

这条 sql 语句锁定了 account 表中所有符合检索条件 (name="Erica") 的记录。本次事务提交之前 (事务提交时会释放事务过程中的锁)，外界无法修改这些记录。Hibernate 的悲观锁，也是基于数据库的锁机制实现。

下面的代码实现了对查询记录的加锁：

[java] [view plaincopyprint?](#)

```
1. String hqlStr = "from TUser as user where user.name='Erica'";  
2. Query query = session.createQuery(hqlStr);  
3. query.setLockMode("user", LockMode.UPGRADE); // 加锁  
4. List userList = query.list(); // 执行查询，获取数据
```

query.setLockMode 对查询语句中，特定别名所对应的记录进行加锁 (我们为 TUser 类指定了一个别名“user”)，这里也就是对返回的所有 user 记录进行加锁。

观察运行期 Hibernate 生成的 SQL 语句：

[sql] [view plaincopyprint?](#)

```
1. select tuser0_.id as id, tuser0_.name as name, tuser0_.group_id  
2. as group_id, tuser0_.user_type as user_type, tuser0_.sex as sex  
3. from t_user tuser0_ where (tuser0_.name='Erica') for update
```


这里 Hibernate 通过使用数据库的 `for update` 子句实现了悲观锁机制。

Hibernate 的加锁模式有：

LockMode.NONE ： 无锁机制。

LockMode.WRITE ：Hibernate 在 Insert 和 Update 记录的时候会自动获取。

LockMode.READ ： Hibernate 在读取记录的时候会自动获取。

以上这三种锁机制一般由 Hibernate 内部使用，如 Hibernate 为了保证 Update 过程中对象不会被外界修改，会在 `save` 方法实现中自动为目标对象加上 `WRITE` 锁。

LockMode.UPGRADE ： 利用数据库的 `for update` 子句加锁。

LockMode. UPGRADE_NOWAIT ： Oracle 的特定实现，利用 Oracle 的 `for update nowait` 子句实现加锁。

上面这两种锁机制是我们在应用层较为常用的，加锁一般通过以下方法实现：

Criteria.setLockMode

Query.setLockMode

Session.lock

注意，只有在查询开始之前（也就是 Hibernate 生成 SQL 之前）设定加锁，才会真正通过数据库的锁机制进行加锁处理，否则，数据已经通过不包含 `for update` 子句的 `Select SQL` 加载进来，所谓数据库加锁也就无从谈起。

在 **Hibernate** 使用悲观锁十分容易，但实际应用中悲观锁是很少被使用的，因为它大大限制了并发性，并且利用数据库底层来维护锁，这样大大降低了应用程序的效率。

下面我们来看一下 **hibernateAPI** 中提供的两个 **get** 方法：

Get (Classclazz, Serializable id, LockMode lockMode)

Get (Classclazz, Serializable id, LockOptions lockOptions)

可以看到 **get** 方法第三个参数"**lockMode**"或"**lockOptions**"，注意在 **Hibernate3.6** 以上的版本中"**LockMode**"已经不建议使用。方法的第三个参数就是用来设置悲观锁的，使用第三个参数之后，我们每次发送的 **SQL** 语句都会加上"**for update**"用于告诉数据库锁定相关数据。**LockMode** 参数选择 **UPGRADE** 选项，就会开启悲观锁。

乐观锁 (**Optimistic Locking**)

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本 (**Version**) 记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 "**version**" 字段来实现。

乐观锁的工作原理：读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前

版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

Hibernate 为乐观锁提供了 3 中实现：

1. 基于 **version**

2. 基于 **timestamp**

3. 为遗留项目添加添加乐观锁

配置基于 **version** 的乐观锁：

[\[html\] view plaincopyprint?](#)

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.
   0//EN""http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
3.
4. <hibernate-mapping>
5.   <classnameclassname="com.bzu.hibernate.pojos.People" table="people">
6.     <idnameidname="id" type="string">
7.       <columnnamecolumnname="id"></column>
8.       <generatorclassgeneratorclass="uuid"></generator>
9.     </id>
10.
11.     <!--version 标签用于指定表示版本号的字段信息-->
12.     <versionnameversionname="version" column="version" type="integer"></ver
        sion>
13.
14.     <propertynamepropertyname="name" column="name" type="string"></prope
        rty>
15.
16.   </class>
17. </hibernate-mapping>
```

注：不要忘记在实体类添加属性 **version**

配置基于 timestamp 的乐观锁：

[html] [view plaincopyprint?](#)

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.
   0//EN""http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
3.
4. <hibernate-mapping>
5.   <classnameclassname="com.suxiaolei.hibernate.pojos.People" table="people">
6.
7.     <id name="id" type="string">
8.       <column name="id"></column>
9.       <generator class="uuid"></generator>
10.    </id>
11.    <!--timestamp 标签用于指定表示版本号的信息-->
12.    <timestamp name="updateDate" column="updateDate"></timestamp>
13.
14.    <propertynamepropertyname="name" column="name" type="string"></prope
    rty>
15.
16.
17.  </class>
18.</hibernate-mapping>
```

下面我们就模拟多个 session，基于 version 的来进行一下测试：



[java] [view plaincopyprint?](#)

```
1. /*
2.     * 模拟多个 session 操作 student 数据表
```

```

3.      */
4.
5.      Session session1=sessionFactory.openSession();
6.      Session session2=sessionFactory.openSession();
7.      Student stu1=(Student)session1.createQuery("from Student s where s.name='
tom11").uniqueResult();
8.      Student stu2=(Student)session2.createQuery("from Student s where s.name='
tom11").uniqueResult();
9.
10.     //这时候，两个版本号是相同的
11.     System.out.println("v1="+stu1.getVersion()+"--v2="+stu2.getVersion());
12.
13.     Transaction tx1=session1.beginTransaction();
14.     stu1.setName("session1");
15.     tx1.commit();
16.     //这时候，两个版本号是不同的，其中一个的版本号递增了
17.     System.out.println("v1="+stu1.getVersion()+"--v2="+stu2.getVersion());
18.
19.     Transaction tx2=session2.beginTransaction();
20.     stu2.setName("session2");
21.     tx2.commit();

```

运行结果：

Hibernate: insert into studentVersion (ver, name,id) values (?, ?, ?)

Hibernate: select student0_.id as id0_, student0_.ver as ver0_,
student0_.name as name0_ from studentVersion student0_ where
student0_.name='tom11'

Hibernate: select student0_.id as id0_, student0_.ver as ver0_,
student0_.name as name0_ from studentVersion student0_ where
student0_.name='tom11'

v1=0--v2=0

Hibernate: update studentVersion set ver=?, name=? where id=? and ver=?

v1=1--v2=0

Hibernate: update studentVersion set ver=?, name=? where id=? and ver=?

Exception in thread "main" org.hibernate.StaleObjectStateException:Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect): [Version.Student#4028818316cd6b460116cd6b50830001]

可以看到，第二个“用户”**session2** 修改数据时候，记录的版本号已经被 **session1** 更新过了，所以抛出了红色的异常，我们可以在实际应用中处理这个异常，例如在处理中重新读取数据库中的数据，同时将目前的数据与数据库中的数据展示出来，让使用者有机会比较一下，或者设计程序自动读取新的数据

（七十七）细谈 Hibernate （十九）Hibernate 二级缓存详解

与 Session 相对的是，SessionFactory 也提供了相应的缓存机制。

SessionFactory 缓存可以依据功能和目的的不同而划分为内置缓存和外置缓存。**SessionFactory 的内置缓存中存放了映射元数据和预定义 SQL 语句**，映射元数据是映射文件中数据的副本，而预定义 SQL 语句是在 Hibernate 初始化阶段根据映射元数据推导出来的。**SessionFactory 的内置缓存是只读的，应用程序不能修改缓存中的映射元数据和预定义 SQL 语句**，因此 SessionFactory 不需要进行内置缓存与映射文件的同步。

SessionFactory 的外置缓存是一个可配置的插件。在默认情况下，SessionFactory 不会启用这个插件。**外置缓存的数据是数据库数据的副本，外置缓存的介质可以是内存或者硬盘**。SessionFactory 的外置缓存也被称为 Hibernate 的二级缓存。由于 Hibernate 的二级缓存是作用在 SessionFactory 范围内的，因而它比一级缓存的范围更广，可以被所有的 Session 对象所共享。

二级缓存的适用范围

Hibernate 的二级缓存作为一个可插入的组件在使用的时候也是可以进行配置的，但并不是所有的对象都适合放在二级缓存中。

在通常情况下会将具有以下特征的数据放入到二级缓存中：

- 很少被修改的数据。
- 不是很重要的数据，允许出现偶尔并发的数据。
- 不会被并发访问的数据。
- 参考数据。

在这里特别要注意的是对放入缓存中的数据不能有第三方的应用对数据进行更改（其中也包括在自己程序中使用其他方式进行数据的修改，例如，JDBC），因为那样 **Hibernate** 将不会知道数据已经被修改，也就无法保证缓存中的数据与数据库中数据的一致性。

二级缓存组件

在默认情况下，**Hibernate** 会使用 **EHCache** 作为二级缓存组件。但是，可以通过设置 `hibernate.cache.provider_class` 属性，指定其他的缓存策略，该缓存策略必须实现 `org.hibernate.cache.CacheProvider` 接口。通过实现 `org.hibernate.cache.CacheProvider` 接口可以提供对不同二级缓存组件的支持。

Hibernate 内置支持的二级缓存组件如表 14.1 所示。

组件	Provider 类	类型	集群
Hashtable	<code>org.hibernate.cache.HashtableCacheProvider</code>	内存	不支持
EHCache	<code>org.hibernate.cache.EhCacheProvider</code>	内存，硬盘	不支持
OSCache	<code>org.hibernate.cache.OSCacheProvider</code>	内存，硬盘	不支持
SwarmCache	<code>org.hibernate.cache.SwarmCacheProvider</code>	集群	支持

下面我们就使用 EhCache 配置二级缓存为例，详细介绍一下二级缓存的配置方法：

- 1) 首先你要把 ehcache-1.2.3.jar 加入到当前应用的 classpath 中。
- 2) 然后设置 EhCache，建立配置文件 ehcache.xml，默认的位置在 class-path，可以放到你的 src 目录下，具体 ehcache.xml 文件内容如下：

[html] [view plaincopyprint?](#)

```

1. <?xmlversionxmlversion=      encoding=      ?>
2. <ehcache>
3. <diskStorepathdiskStorepath=      />
4. <defaultCache
5.   maxElementsInMemory=      <!-- 缓存最大数目 -->
6.   eternal=      <!-- 缓存是否持久 -->
7.
8.   overflowToDisk=      <!-- 是否保存到磁盘，当系统当机时-->
9.
10.  timeToIdleSeconds=      <!-- 当缓存闲置 n 秒后销毁 -->
11.  timeToLiveSeconds=      <!-- 当缓存存活 n 秒后销毁-->
12.  diskPersistent=
13.  diskExpiryThreadIntervalSeconds=      />
14.
15.
16.
17. <cache name="com.bzu.hibernate.Student"      ="200"
18.
19.   eternal=
20.
21.   timeToIdleSeconds=
22.

```

```

23.     timeToLiveSeconds=
24.
25.     overflowToDisk=      />
26.
27.
28. </ehcache>

```

3) 配置完上边之后我们就要在 Hibernate 配置文件中设置，具体配置如下：

[html] view plaincopyprint?

```

1. <!-- 设置 Hibernate 的缓存接口类，这个类在 Hibernate 包中 -->
2. <propertynamepropertyname=                >org.hibernate.ca
    che.EhCacheProvider</property>
3. <!-- 是否使用查询缓存 -->
4.
5. <propertynamepropertyname=                >true</proper
    ty>

```

如果使用 spring 调用 Hibernate 的 sessionFactory 的话，这样设置：

[html] view plaincopyprint?

```

1. <!--HibernateSession 工厂管理 -->
2.
3. <beanidbeanid="sessionFactory"           ="org.springframework.orm.hibernate3.Loc
    alSessionFactoryBean">
4. <propertynamepropertyname=                >
5.
6. <!--其他属性省略。。。-->
7. <props>

```

```

8. <propkeypropkey=                                >org.hibernate.connection.C3P0C
   onnectionProvider</prop>
9. <propkeypropkey=                                >true</prop>
10.<propkeypropkey=                                >org.hibernate.cache.EhCach
    eProvider</prop>
11.</props>
12.</property>
13.<propertynamepropertyname=                        >
14.</list>
15.<value>/WEB-INF/classes/cn/rmic/manager/hibernate/</value>
16.</list>
17.</property>
18.</bean>

```

注意：如果不设置“查询缓存”，那么 **hibernate** 只会缓存使用 **load()**方法获得的单个持久化对象，如果想缓存使用 **findall()**、**list()**、**Iterator()**、**createCriteria()**、**createQuery()**等方法获得的数据结果集的话，就需要设置 **hibernate.cache.use_query_cache true** 才行

如果需要“查询缓存”，还需要在使用 **Query** 或 **Criteria()**时设置其 **setCacheable(true);**

配置完上边通用的配置之后，我们接下来要看怎么对特定的实体进行配置二级缓存了，这里的配置其实也很简单，就在对应的 **hbm** 文件中配置

<cacheusage="read-write"/>就 Ok 了。我们以一个一对多的配置文件来看一下具体的 **hbm** 的配置：

多的一端：

[html] [view plaincopyprint?](#)

```

1. <hibernate-mapping>
2.
3.     <classnameclassname=           table=           >
4.
5.     <!--注意：这一句需要紧跟在 class 标签下面，其他位置无效。 -->
6.
7.     <cacheusagecacheusage=           region=
8.         />
9.     <id name=           column="id"           ="string">
10.
11.         <generatorclassgeneratorclass=           ></generator>
12.
13.     </id>
14.
15.     <property name=           column="name"           ="string"></property>
16.
17.     <property name=           column="cardId"           ="string"></property>
18.
19.     <property name=           column="age"           ="int"></property>
20.
21.     <many-to-one name="team"           ="com.bzu.hibernate.Team"column=
22.         ></many-to-one>
23.
24. </class>
25. </hibernate-mapping>

```

一的一端配置：

[html] view plaincopyprint?

```

1. <set name=           inverse="true"           ="select" lazy="true"           ="save-upd
2.     ate">

```

```

2.
3. <!--注意：这一句需要紧跟在 class 标签下面，其他位置无效。 -->
4.
5.     <cacheusagecacheusage=           region=
        />
6.
7.     <keycolumnkeycolumn=           ></key>
8.
9.     <one-to-manyclassone-to-manyclass=           />
10.
11. </set>

```

hbm 文件查找 cache 方法名的策略： 如果不指定 hbm 文件中的 `region="ehcache.xml 中的 name 的属性值"`，则使用 `name` 名为 `com.ouou.model.Videos` 的 `cache`，如果不存在与类名匹配的 `cache` 名称，则用 `defaultCache`。

选择缓存策略依据：

`<cache usage="transactional|read-write|nonstrict-read-write|read-only"/>`
 ehcache 不支持 **transactional**，其他三种可以支持。

read-only: 无需修改， 那么就可以对其进行只读 缓存，注意，在此策略下，如果直接修改数据库，即使能够看到前台显示效果，但是将对象修改至 `cache` 中会报 `error`，`cache` 不会发生作用。另：删除记录会报错，因为不能在 `read-only` 模式的对象从 `cache` 中删除。

read-write: 需要更新数据，那么使用读/写缓存比较合适，前提：数据库不可以为 `serializable transaction isolation level`

（序列化事务隔离级别）

nonstrict-read-write: 只偶尔需要更新数据（也就是说两个事务同时更新同一记录的情况很不常见），也不需要十分严格的事务隔离，那么比较适合使用非严格读/写缓存策略。

注：在 Spring 托管的 Hibernate 中使用二级缓存

1. 在 spring 的配置文件中，hibernate 部分加入 xml 代码

`org.hibernate.cache.EhCacheProvider true`

2.为 HBM 表设置 cache 策略 xml 代码

3.在 DAO 中，调用 find 方法查询之前，设置使用缓存 Java 代码

`getHibernateTemplate().setCacheQueries(true);`

（七十八）细谈 Spring（七）spring 之 JDBC 访问数据库及配置详解

利用 spring 访问数据库是我们 ssh 程序中必不可少的步骤，在没有 hibernate 之前，我们一般都用 jdbc 访问数据库，所以用 jdbc 访问数据库必不可少的要进行一些配置，spring 中为我们提供了访问数据库的数据源配置，配置完以后我们就可以很容易的利用 jdbc 对数据库进行访问了。下面我们就具体来看一下 spring 所支持的集中 jdbc 数据源的配置：

在 Spring 的配置文件中，关于 dataSource 的配置，就我们常用的方法大致可以有三种：

1、一般的配置方法，直接在配置中指定其值。具体的例子我们参照 Mysql 的配置如下：

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                ><bean id=                class=
2. <property name=                >
3. com.mysql.jdbc.Driver
4. </property>
5. <property name=                >
6. dbc:mysql://localhost:3306/dataBase
7. </property>
8. <property name=                value=                ></property>
9. <property name=                value=                ></property>
10. </bean></SPAN>
```

2.、通过读取文件信息资源，其原理与方法一相同。示例：

[html] view plaincopyprint?

```
1. <SPAN style=                ><bean id=                class=
                                >
2. <property name=                >
3. <value>/WEB-INF/files.properties</value> <!-- 指定文件路径 -->
4. </property>
5. </bean>
6. <bean id=                class=
                                >
7. <property name=                >
8. <value>${driverClassName}</value> <!-- 这里的值要通过${}进行转义，其
    driverClassName 参数要在上面的文件中指定 -->
9. </property>
10. <property name=                >
11. <value>${url}</value>
12. </property>
13. <property name=                value=                ></property>
14. <property name=                value=                ></property>
15. </bean>
16. 3、通过数据连接池。在此我们只需指定 jndiName 的值为服务器中配置的数据连接
    池的 JNDI 名称即可。
17. <bean id=                class=
                                >
18. <property name=                value=                ></property>
19. </bean></SPAN>
```

在上述方法配置成功之后，我们可以通过 JdbcTemplate 把 dataSource 注入到 JdbcTemplate 里面

[html] [view plaincopyprint?](#)

```
1. <SPAN style=                > <bean id=                class=
                                >
2. <property name=            >
3. <ref bean=                />
4. </property>
5. </bean></SPAN>
```

配置完这些之后我们就可以利用 **JdbcTemplate** 来访问数据库了。利用 **JdbcTemplate** 访问数据库要比一般的 **jdbc** 访问数据库方便的多，也简单的多，直接调用相关的访问就 OK 了，也不用管什么关闭和打开链接。下面我们就以一个保存用户实例来简单看一下 **JdbcTemplate** 的基本用法：

[java] [view plaincopyprint?](#)

```
1. <SPAN style="FONT-SIZE: 18px">public class UserDao {
2.     private JdbcTemplate jdbcTemplate;
3.     public DataSource getJdbcTemplate() {
4.         return jdbcTemplate;
5.     }
6.     public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
7.         this.jdbcTemplate= jdbcTemplate;
8.     }
9.     public void insertUser(User user) {
10.         String username = user.getUsername();
11.         String password = user.getPassword();
12.         String email = user.getEmail();
13.         jdbcTemplate.update("insert into user(username,password,email) values ("
14.             + username + "," + password + "," + email + ");");
15.     }
16. }
```

17.

我们看一下在 beans.xml 中对 userdao 的基本配置：

[html] [view plaincopyprint?](#)

```
1. <SPAN style=                                ><bean id=                                class
   =                                            >
2.     <property name=                        >
3.         <ref local=                        />
4.     </property>
5. </bean></SPAN>
```

好了，这样就可以利用 spring

（七十九）细谈 Spring（八）spring+hibernate 整合基本详解

由于 Spring 和 Hibernate 处于不同的层次，Spring 关心的是业务逻辑之间的组合关系，Spring 提供了对他们的强大的管理能力，而 Hibernate 完成了 OR 的映射，使开发人员不用再去关心 SQL 语句，直接与对象打交道。将 Hibernate 做完映射之后的对象交给 Spring 来管理是再合适不过的事情了，Spring 也同时提供了对 Hibernate 的 SessionFactory 的集成功能。所以 spring+hibernate 整合对我们实际开发是非常有必要的。Spring 整合 hibernate 有多种方式，我用的只是其中的一种，我这种不需要 hibernate 的配置文件，直接配置我们的 beans.xml 里了。下面我们具体来看一下：

首先我们先把需要的实体类定义出来，我这里定义的是：

[java] [view plaincopyprint?](#)

```
1. <SPAN style="FONT-SIZE: 18px">Users.java:
2. public class Users {
3.     private int id;
4.     private String name;
5.     public int getId() {
6.         return id;
7.     }
8.     public void setId(int id) {
9.         this.id = id;
10.    }
11.    public String getName() {
12.        return name;
13.    }
14.    public void setName(String name) {
15.        this.name = name;
```

```
16.}
17.}</SPAN>
```

这里我们整合 spring+hibernate 主要是来整合的细节，业务逻辑和分层在此就忽略不计了，我直接把所有的东西都写在了 test 类里面了。

下面我们来看一下我们的 beans.xml 的配置，我们先把代码贴出来，然后再下面一点点的深入详解：

Beans.xml:

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                                ><?xml version=          encodi
   ng=          ?>
2. <beans xmlns=
3. xmlns:xsi=                                xmlns:p=
4. xmlns:tx=
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.s
   pringframework.org/schema/beans/spring-beans-2.5.xsd
6. http://www.springframework.org/schema/tx
7.          http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
8. "></SPAN>
```

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=                                >//配置数据库相关信息，配置
   数据源，它将会被注入到 sessionFactory 中
2. <bean id=          class=
3. destroy-method=          >
4. <propertynamepropertyname=          value=
   </>
5. <property name=
6. value=          >
```

```

7. <property name=          value=    ></property>
8. <property name=          value=    ></property>
9. </bean>
10.//构建 sessionFactory 的 bean，为把 sessionFactory 注入到 dao 层构建 bean
11.<bean id=
12.class=
13.lazy-init=    >
14.//注入 datasource，给 sessionFactoryBean 内 setdatasource 提供数据源
15.<property name=          >
16.<ref bean=          />
17.</property>
18.//加载实体类的配置文件
19.<property name=          >
20.<list>
21.<value>com/bzu/test/Users.hbm.xml</value>
22.<value>com/bzu/test/Log.hbm.xml</value>
23.</list>
24.</property>
25.//设置 hibernate 的属性，相当于 hibernate.cfg.xml 中的设置属性
26.<property name=          >
27.<props>
28.<prop key=          >
29.org.hibernate.dialect.SQLServerDialect
30.</prop>
31.<prop key=          >update</prop>
32.<prop key=          >true</prop>
33.</props>
34.</property>
35.</bean>
36.//把 sessionFactory 注入 dao 层
37.<bean id=      class=          lazy-init=    >
38.<property name=          ref=          ></property>
39.</bean>
40.</beans></SPAN>

```

通过上面的注释我想大家应该大体对这个配置文件明白些了吧。其实这个也很好理解，大体思路就是首先写一个 `datasource` 的 bean，这个 bean 主要是为 `hibernate` 提供数据源，大家肯定可以想到这个数据源将会被注入到 `sessionfactory` 里面，因为构建 `sessionfactory` 肯定会需要到这个数据源的信息。下一步我们在写一个 `sessionfactory` 的 bean，这个 bean 将被我们用在注入到 `dao` 层，进行数据库操作，当然，他还需要一些属性进行注入，比如我们刚才写的 `datasource`，除了这个之外我们还需要配置我们的实体，spring 给我们提供了 `hbm` 文件和实体类等多种配置方法。`Hbm` 文件配置我们这样写：

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=
    ><property name=
    >
2. <list>
3. <value>com/bzu/test/Users.hbm.xml</value>
4. <value>com/bzu/test/Log.hbm.xml</value>
5. </list>
6. </property></SPAN>
```

属性值为 `mappingResources`，这个属性值是一个 `list`，我们可以配置他的 `value`，把我们的配置文件一个个的加入进来。如果你没有写配置文件，而是以注解的方式配置的实体类，你当然也可以以实体类的形式加到这里来：你可以以下方式进行配置：把属性的 `name` 设置成 `annotatedClasses`

[\[html\] view plaincopyprint?](#)

```
1. <SPAN style=
    ><property name=
    >
2. <list>
3. <value>com.bzu.model.User</value>
```

4. `<value>com.bzu.model.Log</value>`
5. `</list>`
6. `</property>`

说完配置 `sessionfactory` 的的实体，下一步我们还要配置 `hibernate` 的的一些属性，比如创建数据库表的方式、数据库方言等。设置 `hibernate` 的属性，我们用 `hibernateProperties` 来配置，配置方式，上面代码已经很清楚的写到了，在这就不赘述了。

好了，配置完 `sessionfactory` 这个 bean，下一步我们就要把 `sessionfactory` 注入到我们的 `dao` 层进行数据库操作了。这一步就比较简单了，相信大家能看懂上面的代码。

看完配置文件，接下来我们就要来测试一下来，试试能不能用 `hibernate` 对我们的数据库进行访问了。具体来看一下代码：

[java] [view plaincopyprint?](#)

```
1. <SPAN style="COLOR: #7f0055; FONT-SIZE: 18px">package com.bzu.test;
2. import org.hibernate.Session;
3. import org.hibernate.SessionFactory;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import org.springframework.transaction.annotation.Transactional;
6. public class test {
7.     private static SessionFactory sessionFactory;
8.     public SessionFactory getSessionFactory() {
9.         return sessionFactory;
10.    }
11.    public void setSessionFactory(SessionFactory sessionFactory) {
12.        this.sessionFactory = sessionFactory;
13.    }
14.    public static void main(String[] args) throws Exception {
```

```

15. ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
16. "beans.xml");
17. Session session = sessionFactory.openSession();
18. // ((SessionFactory) ctx.getBean("sessionFactory")).openSession();
19. Users user = new Users();
20. user.setName("nihao");
21. session.save(user);
22. }
23. }</SPAN>

```

大家可以清楚的看到，上面的内容很简单，我们写具体的架构的东西，直接把 sessionFactory 注入到了 test 类了，当然了，我们实际开发应用中不会手动的去加载 beans.xml 的，这里主要是为了测试，测试一下：

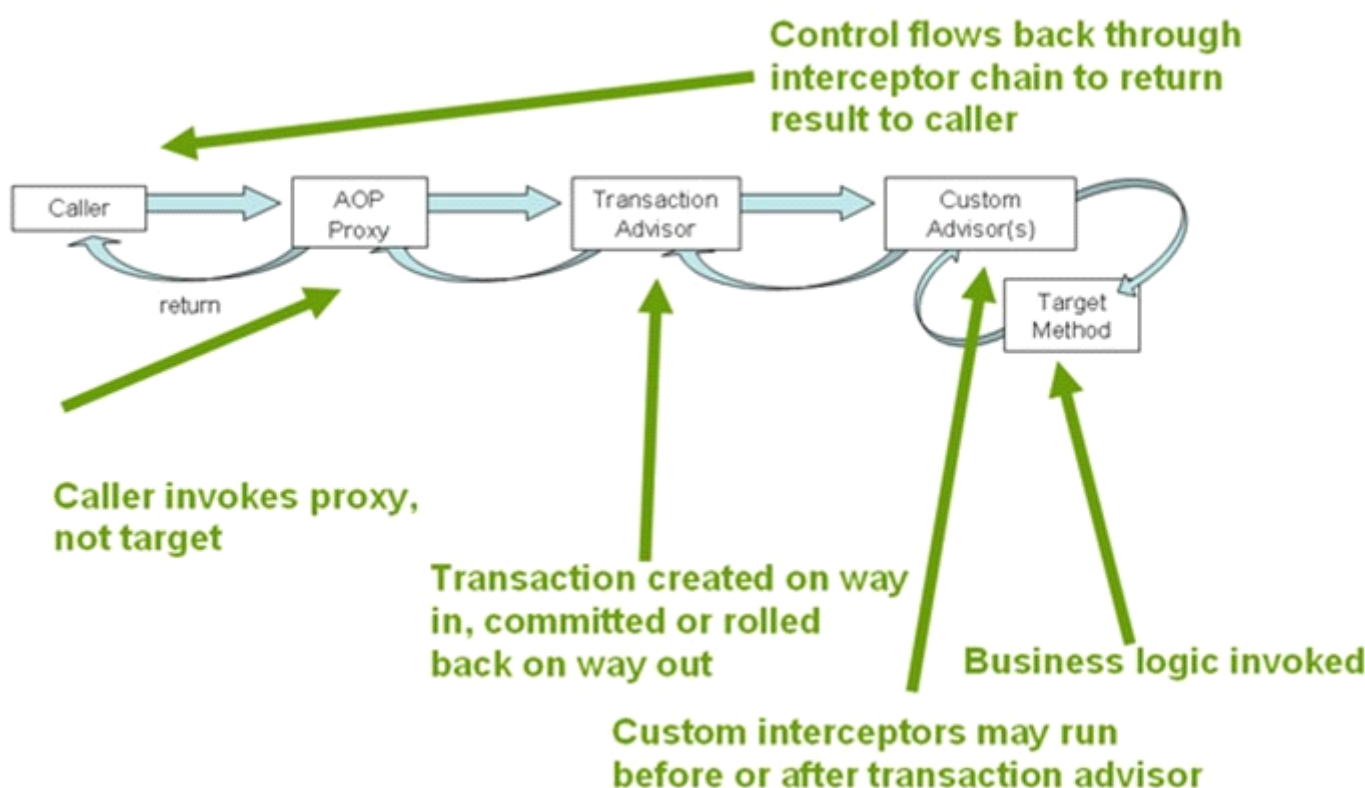
打印出 sql 语句：

Hibernate: insert into users (name) values (?)

好了，基本上我们这个 hibernate+spring 整合的差不多了。但在调试这个程序的时候还是出了点小差错，最后弄了半天才发现错误所在，我刚开始写的时候，是把 main 里面的操作都抽出一个方法了，然后再 main 方法中我是这样调用的 new Test ().save(user)，直接就出事了。异常的原因就是我 new 出来的 test 的对象就不受 spring 容器管理了，所以在调用 save 方法时根本 sessionFactory 就没有被注入进去，所以希望大家注意这一点。

（八十）细谈 Spring（九）spring+hibernate 声明式事务管理详解

声明式事务管理是 spring 对事务管理的最常用的方式，因为这种方式对代码的影响最小，因此也符合非侵入性的轻量级容器的概念。Spring 的事务管理是通过 AOP 的方式来实现的，因为事务方面的代码与 spring 的绑定并以一种样板式结构使用。在理解 spring 声明式事务管理我们首先要理解他是通过 AOP 怎么具体实现的。其中的事务通知由元数据（目前基于 xml 和注解）驱动。代理对象由元数据结合产生一个新的代理对象。他使用一个 PlatformTransactionManager 实现配合 TransactionInterceptor 在方法调用之前实施事务。下面我们就通过一个图来看一下 spring 声明式事务管理的执行过程。



下面我们就以一个 `spring` 官方文档所给的例子来具体看一下用 `xml` 配置方式怎么来实现声明式事务管理：

首先请看下面的接口和它的实现。这个例子的意图是介绍概念：

// 我们想做成事务性的服务接口

[java] [view plaincopyprint?](#)

```
1. package x.y.service;
2. public interface FooService {
3.     Foo getFoo(String fooName);
4.     Foo getFoo(String fooName, String barName);
5.     void insertFoo(Foo foo);
6.     void updateFoo(Foo foo);
7. }
```

// 上述接口的一个实现

[java] [view plaincopyprint?](#)

```
1. package x.y.service;
2. public class DefaultFooService implements FooService {
3.     public Foo getFoo(String fooName) {
4.         throw new UnsupportedOperationException();
5.     }
6.     public Foo getFoo(String fooName, String barName) {
7.         throw new UnsupportedOperationException();
8.     }
9.     public void insertFoo(Foo foo) {
10.        throw new UnsupportedOperationException();
11.    }
12.    public void updateFoo(Foo foo) {
13.        throw new UnsupportedOperationException();
14.    }}
```

首先要解释的是很多同学可能都在考虑这个事务管理到底是放在 dao 层还是放在 service 层呢。这个问题我想大多数童鞋的反应应该都是在 dao 层上吧，刚开始我也是这么想的。但是大家想想，如果我们要进行两个甚至多个 dao 层中的方法操作，并且要求放在同一个事务里时，我们该怎么来管理这个事务呢，这时我们就没办法了。所以我们应该把事务管理放在 service 层中，我们直接在 service 层中调用这两个 dao 层的方法就 oK 了。

下面我们接着往下看，我们假定，FooService 的前两个方法（getFoo(String) 和 getFoo(String, String)）必须执行在只读事务上下文中，其他的方法（insertFoo(Foo)和 updateFoo(Foo)）必须执行在可读写事务上下文中。我们根据这个要求来看一下配置文件，我们刚开始可能看不懂，不用慌，往下我们会一一解释的。

[html] [view plaincopyprint?](#)

1. <!-- from the file 'context.xml' -->
2. <?xml version= encoding= ?>
3. <beans xmlns=
4. xmlns:xsi=
5. xmlns:aop=
6. xmlns:tx=
7. xsi:schemaLocation="
8. http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
9. http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
10. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
- 11.

```

12. <!-- this is the service object that we want to make transactional -->
13. <bean id=          class=          />
14. <!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below)
    -->
15. <tx:advice id=          transaction-manager=          >
16. <!-- the transactional semantics... -->
17. <tx:attributes>
18.   <!-- all methods starting with 'get' are read-only -->
19.   <tx:method name=          read-only=          />
20.   <!-- other methods use the default transaction settings (see below) -->
21.   <tx:method name=          />
22. </tx:attributes>
23. </tx:advice>
24.
25. <!-- ensure that the above transactional advice runs for any execution
26.   of an operation defined by the FooService interface -->
27. <aop:config>
28.   <aop:pointcut id=          expression=
          />
29.   <aop:advisor advice-ref=          pointcut-ref=          />
30. </aop:config>
31.
32. <!-- don't forget the DataSource -->
33. <bean id=          class=          de
    stroy-method=          >
34.   <property name=          value=          />
35.   <property name=          value=          />
36.   <property name=          value=          />
37.   <property name=          value=          />
38. </bean>
39. <!-- similarly, don't forget the PlatformTransactionManager -->
40. <bean id=          class=
          >
41.   <property name=          ref=          />
42. </bean>

```

43. <!-- other <bean/> definitions here -->

44. </beans>

好了，配置一大片，什么东西，我也看不懂，呵呵，没关系，一会大家就明白了，我们先来看一下官方给的解释，然后我在根据我自己的理解给大家通俗的解释一下这里的内容。

我们要把一个服务对象（*'fooService' bean*）做成事务性的。我们想施加的事务语义封装在<tx:advice/>定义中。<tx:advice/> “把所有以 *'get'* 开头的方法看做执行在只读事务上下文中，其余的方法执行在默认语义的事务上下文中”。其中的 *'transaction-manager'* 属性被设置为一个指向 *PlatformTransactionManager bean* 的名字（这里指 *'txManager'*），该 *bean* 将会真正管理事务。配置中最后一段是 <aop:config/> 的定义，它确保由 *'txAdvice' bean* 定义的事务通知在应用中合适的点被执行。首先我们定义了一个切面，它匹配 *FooService* 接口定义的所有操作，我们把该切面叫做 *'fooServiceOperation'*。然后我们用一个通知器（*advisor*）把这个切面与 *'txAdvice'* 绑定在一起，表示当 *'fooServiceOperation'* 执行时，*'txAdvice'* 定义的通知逻辑将被执行。

好了，上面就是官方文档给出的这个配置文件的解释，不知道大家有没有看懂，反正对于初学者我的时候，我是真没看懂，不太容易懂，当然了，大牛们是一定能看懂的。下面我就根据我自己的理解来通俗的讲解一下。

首先我们应该要把服务对象'*fooService*' 声明成一个 **bean**，我们要把一个服务对象（*'fooService' bean*）做成事务性的。我们就应该首先在声明一

个事务管理的建议，用什么来管理，spring 给我们提供了事务封装，这个就封装在了<tx:advice/>中，这个事务建议给我们提供了一个

transaction-manager 属性，用他可以指定我们用谁来管理我们的事务。我们上边的例子用的为一个指向 PlatformTransactionManager bean 的名字（这里指 'txManager'），该 bean 将会真正管理事务。上面用的事务管理类是用的 jdbc 中提供的事务管理，当然这里也可以指定为 hibernate 管理。当然了，不管用那个类来管理我们的事务，都不要忘记了提供我们的 datasource 属性，因为事务管理也需要这里面的信息。我们声明好事务建议，也指定好了具体用哪个类来管理了，下面我们的任务就是要把我们定义好的这些**利用 AOP**

把我们的事务管理织入到我们的业务逻辑里面了。<aop:config/> 的定义，它确保由 'txAdvice' bean 定义的事务通知在应用中合适的点被执行。首先我们定义了一个切面，它匹配 FooService 接口定义的所有操作，我们把该切面叫做 'fooServiceOperation'。<aop:pointcut/> 元素定义是 AspectJ 的切面表示法，上述表示 x.y.service.FooService 包下的任意方法。然后我们用一个通知器（advisor）把这个切面与 'txAdvice' 绑定在一起，表示当 'fooServiceOperation' 执行时，'txAdvice' 定义的通知逻辑将被执行。大体流程就是这样的了。

上面的配置将为'fooService' bean 创建一个代理对象，这个代理对象被装配了事务通知，所以当它的相应方法被调用时，一个事务将被启动、挂起、被标记为只读，或者其它（根据该方法所配置的事务语义）。

我们来看看下面的例子，测试一下上面的配置。

[java] [view plaincopyprint?](#)

```

1. public final class Boot {
2.     public static void main(final String[] args) throws Exception {
3.         ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", B
oot.class);
4.         FooService fooService = (FooService) ctx.getBean("fooService");
5.         fooService.insertFoo (new Foo());
6.     }}

```

运行可以清楚的看到如下结果：

- Invoking rollback for transaction on x.y.service.FooService.insertFoo

due to throwable [java.lang.UnsupportedOperationException]

<tx:advice/> 有关的设置

通过 <tx:advice/> 标签来指定不同的事务性设置。默认的 <tx:advice/> 设置如下：

事务传播设置是 **REQUIRED**

隔离级别是 **DEFAULT**

事务是 读/写

事务超时默认是依赖于事务系统的，或者事务超时没有被支持。

任何 RuntimeException 将触发事务回滚，但是任何 checked Exception 将不触发事务回滚

这些默认的设置当然也是可以被改变的。<tx:advice/> 和 <tx:attributes/> 标签里的 <tx:method/> 各种属性设置总结如下：

Table 9.1. <tx:method/> 有关的设置

属性	是	默认值	描述
----	---	-----	----

否
需
要
?

			与事务属性关联的方法名。通配符 (*)
name	是		可以用来指定一批关联到相同的事务属性的方法。 如: 'get*','handle*','on*Event' 等等。
propagation	不	REQUIRED	事务传播行为
isolation	不	DEFAULT	事务隔离级别
timeout	不	-1	事务超时的时间 (以秒为单位)
read-only	不	false	事务是否只读?
			将被触发进行回滚的 Exception(s); 以逗号
rollback-for	不		分开。 如: 'com.foo.MyBusinessException,ServletException'
			不被触发进行回滚的 Exception(s); 以逗
no-rollback-for	不		号分开。 如: 'com.foo.MyBusinessException,ServletException'

下面我们具体来看一下事务的传播性的几个值:

REQUIRED:业务方法需要在一个容器里运行。如果方法运行时,已经处在一个事务中,那么加入到这个事务,否则自己新建一个新的事务。

NOT_SUPPORTED:声明方法不需要事务。如果方法没有关联到一个事务，容器不会为他开启事务，如果方法在一个事务中被调用，该事务会被挂起，调用结束后，原先的事务会恢复执行。

REQUIRESNEW:不管是否存在事务，该方法总汇为自己发起一个新的事务。如果方法已经运行在一个事务中，则原有事务挂起，新的事务被创建。

MANDATORY:该方法只能在一个已经存在的事务中执行，业务方法不能发起自己的事务。如果在没有事务的环境下被调用，容器抛出例外。

SUPPORTS:该方法在某个事务范围内被调用，则方法成为该事务的一部分。如果方法在该事务范围外被调用，该方法就在没有事务的环境下执行。

NEVER:该方法绝对不能在事务范围内执行。如果在就抛例外。只有该方法没有关联到任何事务，才正常执行。

NESTED:如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按 **REQUIRED** 属性执行。它使用了一个单独的事务，这个事务 拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对 DataSourceTransactionManager 事务管理器起效。

使用 **@Transactional**

除了基于 XML 文件的声明式事务配置外，你也可以采用基于注解式的事务配置方法。直接在 Java 源代码中声明事务语义的做法让事务声明和将受其影响的代码距离更近了，而且一般来说不会有不恰当的耦合的风险，因为，使用事务性的代码几乎总是被部署在事务环境中。

下面的例子很好地演示了 **@Transactional** 注解的易用性，随后解释其中的细节。先看看其中的类定义：

[java] [view plaincopyprint?](#)

```
1. // the service class that we want to make transactional
2. @Transactional
3. public class DefaultFooService implements FooService {
4.     Foo getFoo(String fooName);
5.     Foo getFoo(String fooName, String barName);
6.     void insertFoo(Foo foo);
7.     void updateFoo(Foo foo);
8. }
```

当上述的 POJO 定义在 Spring IoC 容器里时，上述 bean 实例仅仅通过一行 xml 配置就可以使它具有事务性的。如下：

[html] [view plaincopyprint?](#)

```
1. <?xml version=      encoding=      ?>
2. <beans xmlns=
3.     xmlns:xsi=
4.     xmlns:aop=
5.     xmlns:tx=
6.     xsi:schemaLocation="
7.     http://www.springframework.org/schema/beans http://www.springframework.org/
       schema/beans/spring-beans-2.5.xsd
8.     http://www.springframework.org/schema/tx http://www.springframework.org/sch
       ema/tx/spring-tx-2.5.xsd
9.     http://www.springframework.org/schema/aop http://www.springframework.org/s
       chema/aop/spring-aop-2.5.xsd">
10.
11. <bean id=      class=      />
12. <tx:annotation-driven transaction-manager=      />
13. <bean id=      class=
      >
14. <property name=      ref=      />
```

15. `</bean>`

16. `</beans>`

注意： 实际上，如果你用 'transactionManager' 来定义 PlatformTransactionManager bean 的名字的话，你就可以忽略 `<tx:annotation-driven/>` 标签里的 'transaction-manager' 属性。 如果 PlatformTransactionManager bean 你要通过其它名称来注入的话，你必须用 'transaction-manager' 属性来指定它。

在多数情形下，方法的事务设置将被优先执行。在下列情况下，例如： DefaultFooService 类在类的级别上被注解为只读事务，但是，这个类中的 updateFoo(Foo) 方法的 @Transactional 注解的事务设置将优先于类级别注解的事务设置。

[html] [view plaincopyprint?](#)

```
1. @Transactional(readOnly = )
2. public class DefaultFooService implements FooService {
3.     public Foo getFoo(String fooName) {
4.         // do something
5.     }
6.     // these settings have precedence for this method
7.     @Transactional(readOnly = , propagation = .REQUIRES_NEW)
8.     public void updateFoo(Foo foo) {
9.         // do something
10.
11.     }
12. }
```

@Transactional 有关的设置

@Transactional 注解是用来指定接口、类或方法必须拥有事务语义的元数据。 如：“当一个方法开始调用时就开启一个新的只读事务，并停止掉任何现存的事务”。 默认的 @Transactional 设置如下：

事务传播设置是 PROPAGATION_REQUIRED

事务隔离级别是 ISOLATION_DEFAULT

事务是 读/写

事务超时默认是依赖于事务系统的，或者事务超时没有被支持。

任何 RuntimeException 将触发事务回滚，但是任何 checked Exception 将不触发事务回滚

这些默认的设置当然也是可以被改变的。 @Transactional 注解的各种属性设置总结如下：

@Transactional 注解的属性

属性	类型	描述
propagation	枚举型： Propagation	可选的传播性设置
isolation	枚举型： Isolation	可选的隔离性级别（默认值： ISOLATION_DEFAULT）
readOnly	布尔型	读写型事务 vs. 只读型事务
timeout	int 型（以秒为	事务超时

	单位)	
rollbackFor	<p>一组 Class 类的实例，必须是 Throwable 的子类</p> <p>一组异常类，遇到时必须进行回滚。默认情况下 checked exceptions 不进行回滚，仅 unchecked exceptions (即 RuntimeException 的子类) 才进行事务回滚。</p>	
rollbackForClassname	<p>一组 Class 类的名字，必须是 Throwable 的子类</p> <p>一组异常类名，遇到时必须进行回滚</p>	
noRollbackFor	<p>一组 Class 类的实例，必须是 Throwable 的子类</p> <p>一组异常类，遇到时必须不回滚。</p>	
noRollbackForClassname	<p>一组 Class 类的名字，必须是 Throwable 的子类</p> <p>一组异常类，遇到时必须不回滚</p>	

在写代码的时候，不可能对事务的名字有个很清晰的认识，这里的名字是指会在事务监视器（比如 WebLogic 的事务管理器）或者日志输出中显示的名字，对于声明式的事务设置，事务名字总是全限定名+"."+事务通知的方法名。比如 BusinessService 类的 handlePayment(..)方法启动了一个事务，事务的名称是：

`com.foo.BusinessService.handlePayment`

（八十一）细谈 Spring（十）深入源码分析 Spring 之 HibernateTemplate 和 HibernateDaoSupport

spring 提供访问数据库的有三种方式：

HibernateDaoSupport

HibernateTemplate（推荐使用）

jdbcTemplate(我们一般不用)

类所在包：

HibernateTemplate：

org.springframework.orm.hibernate3.HibernateTemplate

HibernateDaoSupport：

org.springframework.orm.hibernate3.support.HibernateDaoSupport

spring 如果想整合 hibernate 的话，首先就应该获得 SessionFactory 这个类，然后再通过获得 session 就可以进行访问数据库了，即 spring 提供的类 HibernateDaoSupport，HibernateTemplate 应该是有 setSessionFactory,在使用的时候注入一下就可以了。HibernateTemplate 类中的方法是 spring 封装了 hibernate 中的方法，在使用完了以后会自动释放 session。而如果使用了 HibernateDaoSupport 的 getSession 方法，就需要配套的用 releaseSession(Session session)或者 session.close 来关闭 session，无法实现自动管理 session。所以很多人都倾向于用 spring

的 `HibernateTemplate` 类或者 `HibernateDaoSupport` 的 `getHibernateTemplate` 方法来实现实现数据库的交互，当然，如果遇到 `hibernatetemplate` 无法实现的功能，可以使用 `HibernateDaoSupport`。

首先我们先来看一下 **`HibernateTemplate`** 类：

首先我们来说一下我们为什么要用 `HibernateTemplate`，其实这个类就是我们平常使用 `hibernate` 进行 `dao` 操作的一个模版，我们不需要那些开头的开启事务、获得 `session`，结尾的提交事务，关闭 `session` 等操作了，这些工作是 `HibernateTemplate` 都给我们封装好了，我们直接调用其 `dao` 的操作方法就可以了，并且他还给我们封装了 `hibernate` 的几乎所有的异常，这样我们在处理异常的时候就不要记住那么多繁琐的异常了。所以我们就叫他是一个 `hibernate` 中 `dao` 操作的模版，他提供的常用方法：

`get` 从数据库相关表中获取一条记录并封装返回一个对象(`Object`)

`load` 作用与 `get` 基本相同，不过只有在对该对象的数据实际调用时，才会去查询数据库

`save` 添加记录

`saveOrUpdate` 判断相应记录是否已存在，据此进行添加或修改记录

`update` 修改记录

`delete` 删除记录

下面我们来看一下 **HibernateTemplate** 的源码来看一下他的具体方法是怎么实现的，其实你观察源码可以发现，他所提供的方法几乎都是一个实现实现的。下面我们就以 **save** 方法来具体看一下：

[java] [view plaincopyprint?](#)

```
1. public Serializable save(final Object entity) throws DataAccessException {  
2.     return (Serializable) executeWithNativeSession(new HibernateCallback() {  
3.         public Object doInHibernate(Session session) throws HibernateException {  
4.             checkWriteOperationAllowed(session);  
5.             return session.save(entity);  
6.         }  
7.     });}
```

我们从源码中可以发现，**HibernateTemplate** 把我们 **hibernate** 的异常都封装成了一个 **DataAccessException**。好了，解释一下上面的代码，上面代码中主要是调用了 **executeWithNativeSession** 这个方法，其实这个方法就是给我们封装好的 **hibernate** 开头和结尾一些列操作，他需要一个参数，这个参数是一个回调的对象，其实这个对象是实现了 **HibernateCallback** 的接口，实现这个接口需要实现这个接口里面的方法 **doInHibernate**，这个方法需要把当前的 **session** 传递过来，其实他就是把他原先模版里获得的 **session** 传过去。然后在 **doInHibernate** 中利用模版中得到的 **session** 进行保存数据。其实我们调用 **save** 的过程就是给他传一个回调对象的过程，我们可以看到，他的回调对象是 **new** 出来的。

如果你还没看懂的话，那大家来看一下下面我们实现自己的
HibernateTemplate，他的思路和 spring 提供的基本是一样的：其中
MyHibernateCallback 是一个简单接口：

[java] [view plaincopyprint?](#)

```
1. import org.hibernate.Session;
2. public class MyHibernateTemplate {
3.     public void executeWithNativeSession(MyHibernateCallback callback) {
4.         Session s = null;
5.         try {
6.             s = getSession();
7.             s.beginTransaction();
8.             callback.doInHibernate(s);
9.             s.getTransaction().commit();
10.        } catch (Exception e) {
11.            s.getTransaction().rollback();
12.        } finally {
13.            //...
14.        }
15.    }
16.    private Session getSession() {
17.        // TODO Auto-generated method stub
18.        return null;
19.    }
20.    public void save(final Object o) {
21.        new MyHibernateTemplate().executeWithNativeSession(new MyHibernateCallbac
            k() {
22.            public void doInHibernate(Session s) {
23.                s.save(o);
24.            }
25.        });
26.    }
27.    }
```

好了，原理我们介绍完了之后，下面我们来看一下具体应用，这个 HibernateTemplate 在我们的程序中怎么用，在上面我们也说过了，这个用法主要是把 sessionFactory 注入给我们的 HibernateTemplate 首先我们来看一下 beans.xml 的配置：

[html] [view plaincopyprint?](#)

```
1. <?xml version=      encoding=      ?>
2. <beans xmlns=
3. xmlns:xsi=
4. xmlns:context=
5. xmlns:aop=
6. xmlns:tx=
7. xsi:schemaLocation="http://www.springframework.org/schema/beans
8.      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
9.      http://www.springframework.org/schema/context
10.     http://www.springframework.org/schema/context/spring-context-2.5.xsd
11.     http://www.springframework.org/schema/aop
12.     http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
13.     http://www.springframework.org/schema/tx
14.     http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
15. <bean id=
16. class=
17. destroy-method=      >
18. <property name=      value=      />
19. <property name=      value=      />
20. <property name=      value=      />
21. <property name=      value=      />
22. </bean>
23. <bean id=
24. class=
    >
```

```

25. <property name=                ref=                />
26. <property name=                >
27. </list>
28. <value>com.bjsxt.model.User</value>
29. <value>com.bjsxt.model.Log</value>
30. </list>
31. </property>
32. <property name=                >
33. <props>
34. <prop key=                >
35. org.hibernate.dialect.MySQLDialect
36. </prop>
37. <prop key=                >true</prop>
38. </props>
39. </property>
40. </bean>
41. <bean id=                class=
                >
42. <property name=                ref=                ></property>
43. </bean>
44. <bean id=                class=                >
45. <property name=                ref=                ></property>
46. </bean>
47. </beans>

```

下一步我们来看一下 `hibernateTemplate` 的使用：

[html] [view plaincopyprint?](#)

```

1. public class UserDaoImpl implements UserDao {
2.     private HibernateTemplate hibernateTemplate;
3.     public HibernateTemplate getHibernateTemplate() {
4.         return hibernateTemplate;
5.     }
6.     public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {

```

```

7. this.hibernateTemplate =          ;
8. }
9. public void save(User user) {
10.hibernateTemplate.save(user);
11.}}

```

这基本上就是我们的 **hibernateTemplate** 原理及使用了，其实他的使用很简单

下面，我们来看一下 **HibernateDaoSupport**:

通过上面我们可以看出，通过 **xml** 注入 **hibernateTemplate**，我们可以想象的到所有 **DAO** 类中都会有 **HibernateTemplate** 的 **bean** 方法，于是上面 **hibernateTemplate** 的 **set**、**get** 的方法和 **xml** 配置会有大量的，于是就出现了代码冗余和重复，我们怎么才能避免这个重复呢，我们很容易应该能想到，把上面注入 **hibernateTemplate** 抽出一个类，然后让我们的 **dao** 类来继承这个类。不过这个类 **Spring** 已经有了，那就是 **HibernateDaoSupport**，除此之外，**HibernateDaoSupport** 也有 **SessionFactory** 的 **bean** 方法，所以我们在用 **HibernateDaoSupport** 的时候同样也要给我们注入 **sessionfactory** 或者 **hibernateTemplate**，在用的时候你会发现 **HibernateDaoSupport** 也给我们提供了 **getHibernateDaoSupport** 方法。

相关配置示例：userdao 继承了 **HibernateDaoSupport**

[html] [view plaincopyprint?](#)

```

1. <bean id=          class=          >
2. <property name=          ref=          ></property>

```

3. `</bean>`

用上面的方法我们可以发现一个问题，我们同样解决不了 xml 配置重复的问题，我们每一个 dao 都要在 xml 注入 sessionFactory 或者 hibernateTemplate，解决这个问题的办法就是我们自己在抽出一个 SuperDao 类，让这个类去继承 HibernateDaoSupport，然后我们给 SuperDao 类去配置，这样的话，我们在我的 dao 类中直接去继承 SuperDao 类就可以了，这样不管有多少 dao 类，只要继承 SuperDao，我们就可以实现我们想要的功能了。

（八十二）细谈 Spring（十一）深入理解 spring+struts2 整合（附源码）

Spring 和 struts2 是我们在项目架构中用的比较多的两个框架，怎么才能把这两个框架用好，怎么来整合是我们掌握运用这两个框架的关键点，下面我们就怎么来整合，从哪来整合，为什么要整合，从这几点来看一下 struts2 和 spring 的整合。下面我们来具体分析一下：

我们一起来想想，如果让 spring 和 struts2 进行整合，我们就希望我们可以在 spring 中直接注入 action，spring 容器初始化的时候就给我们建好了 action，但是事情不像我们想象的那么简单，因为 struts2 的 action 是由 struts2 自己 new 出来的，他不受 spring 的管理，所以无法自动注入。所以 struts 和 spring 的整合的结合点在于，struts2 的 action 不能直接入 service。好了，既然有了问题，spring 或者 struts2 肯定已经为我们把这个问题解决了。struts2 解决这个问题是他给我们提供了一个 struts-spring-plugin 的插件，通过这个插件我们就可以把我们的 struts2 和是 spring 进行整合了。struts-spring-plugin 将会对我们的 action 进行管理，当 spring 需要 action 的时候他就可以向 struts-spring-plugin 来要了。

源码下载：**用力点**

下面我们就具体的来看一下 struts+spring 的整合过程：

1.需要的 jar 包列表：Struts2.1.6 + Spring2.5.6 + Hibernate3.3.2

jar 包名称	所在位置	说明
antlr-2.7.6.jar	hibernate/lib/required	解析 HQL
aspectjrt	spring/lib/aspectj	AOP
aspectjweaver	..	AOP
cglib-nodep-2.1_3.jar	spring/lib/cglib	代理，二进制增强
common-annotations.jar	spring/lib/j2ee	@Resource
commons-collections-3.1.jar	hibernate/lib/required	集合框架
commons-fileupload-1.2.1.jar	struts/lib	struts
commons-io-1.3.2	struts/lib	struts
commons-logging-1.1.1	单独下载，删除 1.0.4(struts/lib)	struts spring
dom4j-1.6.1.jar	hibernate/required	解析 xml
ejb3-persistence	hibernate-annotation/lib	@Entity
freemarker-2.3.13	struts/lib	struts
hibernate3.jar	hibernate	
hibernate-annotations	hibernate-annotation/	
hibernate-common-annotations	hibernate-annotation/lib	
javassist-3.9.0.GA.jar	hibernate/lib/required	hibernate
jta-1.1.jar	..	hibernate transaction
junit4.5		
mysql-ognl-2.6.11.jar	struts/lib	
slf4j-api-1.5.8.jar	hibernate/lib/required	hibernate-log
slf4j-nop-1.5.8.jar	hibernate/lib/required	
spring.jar	spring/dist	
struts2-core-2.1.6.jar	struts/lib	
xwork-2.1.2.jar	struts/lib	struts2
commons-dbcp	spring/lib/jakarta-com	

```

                                mons
commons-pool.jar                ..
struts2-spring-plugin-2.1.6. struts/lib
jar

```

2.配置好 jar 包以后，如果我们想在服务器一启动就可以让 spring 容器自动去加载我们在配置文件中配置的 bean，那么我们要在 web.xml 中去配置一个监听器，这个监听器的作用是监听我们的 application，一旦我们的项目启动就触发了监听器，我们来看一下这个监听器的配置：

[html] [view plaincopyprint?](#)

1. `<listener>`
2. `<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>`
3. `<!-- default: /WEB-INF/applicationContext.xml -->`
4. `</listener>`

如果你的配置文件不想放在默认的位置，而是自己去指定位置，那么我们要在 web.xml 中再次配置如下：

[html] [view plaincopyprint?](#)

1. `<context-param>`
2. `<param-name>contextConfigLocation</param-name>`
3. //这种配置可以指定多个配置文件，因为 spring 的配置文件可以分开写成好几个
4. `<!-- <param-value>/WEB-INF/applicationContext-*.xml,classpath*:applicationContext-*.xml</param-value> -->`
5. //指定 spring 配置文件的位置 classpath 下的 beans.xml
6. `<param-value>classpath:beans.xml</param-value>`

7. `</context-param>`

加载完上面的 **bean** 之后，我们就要考虑去管理我们的 **action** 了，我们应该让 **spring** 去找 **struts** 去要相应的 **action**，把 **action** 实例化为响应的 **bean**，这时我们就需要我们上边所提到的 **struts-spring-plugin** 这个 **jar** 包了。加上这个 **jar** 包之后我们就可以让 **spring** 来管理我们的 **action** 了。在 **struts-spring-plugin** 中有一个 **struts--plugin**。Xml 文件。下面我们来看一下这个文件中的配置执行的具体过程：

[html] [view plaincopyprint?](#)

```
1. <struts>
2.   <bean type=                name=                class=
                                   />
3.
4.   <!-- Make the Spring object factory the automatic default -->
5.   <constant name=                value=                />
6.   <package name=                >
7.     <interceptors>
8.       <interceptor name=                class=
                                   />
9.       <interceptor name=                class=
                                   />
10.    </interceptors>
11.  </package>
12.</struts>
```

关键是这

个 `<constant name="struts.objectFactory" value="spring" />`, 这句配置就指明了我们产生 **struts** 对象的工厂交给 **spring** 来产生, 我们来看一下具体步骤:

struts2 一起启动就会去加载配置文件, 其中包括 **struts-plugin.xml** 读取顺序:

struts 的读常量:

struts-default.xml

struts-plugin.xml

struts.xml

struts.properties

web.xml

struts-plugin.xml 指明了我们产生对象的工厂交给 **spring** 来完成, 当执行到 **web.xml** 时, 由于 **spring** 容器的监听器, 这时 **spring** 容器就开始启动, **spring** 启动之后会 **web.xml** 去找相应的配置, 在 **web.xml** 中可以找到 **spring** 中的配置文件 **beans.xml**, 然后去初始化所有的 **bean**。

spring 去加载 **beans.xml** 的时候会自动把所有的 **bean** 初始化, 并且放在自己的容器里。与此同时, **struts2** 也有自己的 **bean** 容器, 这个容器是 **struts-plugin** 提供的, 他会把所有的 **action** 加载都加载到自己的容器里。然后根据 **action** 的属性名字自动去 **spring** 去找名字和 **action** 属性相同的 **bean** 直接注入到 **action** 中, 也就是说。我们在 **action** 中其实不用配置注入的东西, **struts** 容器会自动给我们注入。但还是要提供相应的 **set**、**get** 方法。并且名字要约定好, **action** 属性和 **spring** 中的 **bean** 的 **id** 要一样。但不要忘记, **action** 的 **scope** 设置成 **prototype**

下面我们来看一下具体的示例代码：

[java] [view plaincopyprint?](#)

```
1. package com.bzu.action;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. import com.bzu.entity.Student;
4. import com.bzu.service.StudentService;
5. import com.opensymphony.xwork2.ActionSupport;
6. public class StudentAction extends ActionSupport {
7.     private Student student;
8.     private StudentService service;
9.     @Override
10. public String execute() throws Exception {
11. // TODO Auto-generated method stub
12. service.login(student);
13. return SUCCESS;
14. }
15. public StudentService getService() {
16. return service;
17. }
18. public Student getStudent() {
19. return student;
20. }
21. public void setStudent(Student student) {
22. this.student = student;
23. }
24. public void setService(StudentService service) {
25. this.service = service;
26. }
27. //public static void main(String[] args) {
28. //ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContex
    t(
29. //""beans.xml");
30. //
```

```

31.//StudentAction action = (StudentAction) context.getBean("StudentAction");
32.//
33.//System.out.println(action == null);
34.//
35.//StudentService ser = action.getService();
36.//
37.//ser.login(new Student("ss", "ss"));
38.//
39.//}

```

需要注意的是，**spring** 和 **struts2** 整合的时候有一个属性 `struts.objectFactory.spring.autoware`，也就是说 `struts` 属性的自动装配类型，他的默认值是 `name`，也就是说 `struts` 中的 `action` 中的属性不需要配置，他默认的去 `beans.xml` 中去找名字相同的，应该注意的是，在给一些属性起名字的时候不要和 `spring` 中配置的 `action` 的 `name` 属性相同，否则会报异常

下面我们来看一下 `struts.xml` 和 `beans.xml` 的相关配置：

Struts.xml:

[html] [view plaincopyprint?](#)

```

1. <struts>
2. <package name=           extends=           >
3. <action name=           class=           >
4. <result name=           >/Success.jsp</result>
5. <result name=           >/Fail.jsp</result>
6. </action>
7. </package>
8. </struts>

```

Beans.xml

[html] view plaincopyprint?

```
1. <bean id=          class=          ></bean>
2. <bean id=          class=          >
3. <property name=      ref=          ></property>
4. </bean>
5. <!--这里完全可以不写-->
6. <!--
7. <bean id=          class=          scope=
   >
8. <property name=      ref=          ></property>
9. </bean>
10.-->
```

上面的示例是用的 struts2 的容器来产生 action，spring 需要的时候要去 struts 容器里面去找 action，然后再初始化 bean，其实我们还有一种方法产生 action，那就是让 spring 容器去帮我们产生 action，这样我们产生 action 的时候就可以去 spring 容器里面去找了，具体应该是在 spring 配置文件 beans.xml 中把对应的 action 配置成 bean，然后再 struts.xml 中配置 action 的时候，action 对应的 class 就不再是配置成该 action 对应的类了，而是配置这个 action 对应的 spring 容器的 bean 的 id 属性，这样 action 初始化的时候就会去 spring 容器里面去找了。但是这样配置的话，我们的 action 属性就必须配置了，因为 spring 来产生 action 后，struts 容器就不会在自动去给我们注入属性了。如果不配置属性的话会产生异常，下面我们来看一下具体配置情况：

Action 的代码还是以前的代码，没有改变，这里就不再重复写了，下面我们来看一下 struts.xml 的配置：

Struts.xml

[\[html\] view plaincopyprint?](#)

```
1. <struts>
2. <package name=           extends=           >
3. <action name=           class=           >
4. <result name=           >/Success.jsp</result>
5. <result name=           >/Fail.jsp</result>
6. </action>
7. </package>
8. </struts>
```

上面的 class 对应的是下面 action 中 bean 的 id 属性

Beans.xml

[\[html\] view plaincopyprint?](#)

```
1. <PRE class=           name=           ><bean id=           class=
           ></bean>
2. <bean id=           class=           >
3. <property name=           ref=           ></property>
4. </bean>
5. <!--现在这里必须写了，不写会报错-->
6. <bean id=           class=           scope=           >
7. <property name=           ref=           ></property>
8. </bean></PRE><BR>
9. <BR>
10.<PRE></PRE>
11.<P></P>
12.<PRE></PRE>
13.<P></P>
14.<P></P>
```


15. <P></P>

16. <P> OK<SPAN style=

> ,

struts2+spring<

SPAN style= >整合讲到这就基本完了，当然我说的也不是很

全面，希望大家见谅，希望大家能提出宝贵意见 </P>

17. <P></P>

18. <PRE></PRE>

19. <PRE></PRE>

（八十三）细谈 Spring（十二）OpenSessionInView 详解及用法

首先我们来看一下什么是 OpenSessionInView？

在 `hibernate` 中使用 `load` 方法时，并未把数据真正获取时就关闭了 `session`，当我们真正想获取数据时会迫使 `load` 加载数据，而此时 `session` 已关闭，所以就会出现异常。比较典型的是在 MVC 模式中，我们在 M 层调用持久层获取数据时(持久层用的是 `load` 方法加载数据)，当这一调用结束时，`session` 随之关闭，而我们希望在 V 层使用这些数据，这时才会迫使 `load` 加载数据，我们就希望这时的 `session` 是 `open` 着的，这就是所谓的 Open Session In view。我们可以用 `filter` 来达到此目的。这段话引至于百度百科，但确实很好的说明了 OpenSessionInView 这个过滤器的作用。OpenSessionInViewFilter 是 Spring 提供的一个针对 *Hibernate* 的一个支持类，其主要意思是在发起一个页面请求时打开 *Hibernate* 的 *Session*，一直保持这个 *Session*，直到这个请求结束，具体是通过一个 *Filter* 来实现的。由于 *Hibernate* 引入了 *Lazy Load* 特性，使得脱离 *Hibernate* 的 *Session* 周期的对象如果再想通过 *getter* 方法取到其关联对象的值，*Hibernate* 会抛出一个 *LazyLoad* 的 *Exception*。所以为了解决这个问题，Spring 引入了这个 *Filter*，使得 *Hibernate* 的 *Session* 的生命周期变长。

首先分析一下它的源码，可以发现，它所实现的功能其实比较简单：

[java] [view plain](#)[copy](#)[print?](#)

```

1. SessionFactory sessionFactory = lookupSessionFactory(request);
2. Session session = null;
3. boolean participate = false;
4.
5. if (isSingleSession()) {
6.     // single session mode
7.     if (TransactionSynchronizationManager.hasResource(sessionFactory)) {
8.         // Do not modify the Session: just set the participate flag.
9.         participate = true;
10.    } else {
11.        logger.debug("Opening single Hibernate Session in OpenSessionInViewFilter");

12.        session = getSession(sessionFactory);
13.        TransactionSynchronizationManager.bindResource(sessionFactory, new SessionHolder(session));
14.    }
15.} else {
16.    // deferred close mode
17.    if (SessionFactoryUtils.isDeferredCloseActive(sessionFactory)) {
18.        // Do not modify deferred close: just set the participate flag.
19.        participate = true;
20.    } else {
21.        SessionFactoryUtils.initDeferredClose(sessionFactory);
22.    }
23.}
24.
25. try {
26.     filterChain.doFilter(request, response);
27. } finally {
28.     if (!participate) {
29.         if (isSingleSession()) {
30.             // single session mode
31.             TransactionSynchronizationManager.unbindResource(sessionFactory);
32.             logger.debug("Closing single Hibernate Session in OpenSessionInViewFilter")
;

```

```

33.     closeSession(session, sessionFactory);
34. }else {
35.     // deferred close mode
36.     SessionFactoryUtils.processDeferredClose(sessionFactory);
37. }
38.}
39.}

```

在上述代码中，首先获得 *SessionFactory*，然后通过 *SessionFactory* 获得一个 *Session*。然后执行真正的 *Action* 代码，最后根据情况将 *Hibernate* 的 *Session* 进行关闭。整个思路比较清晰。

下面我们来看一下他的具体配置，其实很简单，直接在 `web.xml` 中把他这个 filter 配置上就 ok 了：

[html] [view plaincopyprint?](#)

```

1. <filter>
2. <filter-name>openSessionInView</filter-name>
3. <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFil
   ter</filter-class>
4. <init-param>
5. <param-name>sessionFactoryBeanName</param-name>
6. <param-value>sf</param-value>
7. </init-param>
8. </filter>
9. <filter-mapping>
10. <filter-name>openSessionInView</filter-name>
11. <url-pattern>/*</url-pattern>
12. </filter-mapping>

```

在上面配置中我们要注意以下几点：

1、这个 filter 一定要配置在 struts 的过滤器的前面，因为过滤器是“先进后出”原则，如果你配置在 struts 的后面，你的 openSessionInView 过滤器都执行完了，怎么在去在管理 action 的转向页面啊。

2、Opensessioninview 也需要 sessionFactory 的 bean 的注入，他默认的去找 bean 的 id 为 sessionFactory 的 bean，如果 sessionFactory 的 bean 的 id 不是这个名字的话，要记得给这个过滤器配置一个参数，参数名为 sessionFactoryBeanName，把他的 value 设置为你的 sessionFactory 的 bean 的 id 值。

3、在用 opensessioninview 的时候一定要注意，如果你不配置 transaction 的话，在用 opensessioninview 时，他默认的把事务配置为 only-read 只读，这样的话，如果你进行增删改的时候，他就会报一个在只读事务中不能进行增删改的操作。如果把 opensessioninview 去掉，他默认的事务的开始边界就锁定在 dao 层操作上，dao 层 hibernateempt 提供了事务的开始和提交

OpenSessionInView 的副作用

了解了上面几个问题之后，那么也就可以大概知道

OpenSessionInView 的副作用 – 资源占用严重，配置不当，影响系统性能。

使用 OpenSessionInView 后，在 request 发出和 response 返回的流程中，如果有任何一步被阻塞，那在这期间 connection 就会被一直占用而不释放。比如页面较大，显示需要时间 或者 网速太慢，服务器与用户间传输的时间太长，这都会导致资源占用，最直接的表现就是连接池连接不够用，而导致最终服务器无法提供服务。

（八十四）深入浅出 Ajax

Ajax（Asynchronous JavaScript and XML），异步的 JavaScript 与 XML。

所谓同步，就是在进行一个操作之前必须要等到上一个操作返回操作结果才能进行这个操作，而**异步**则是在进行一个操作时可以不受上一个操作的影响，上一个操作是否返回操作都可以执行这个操作，各个操作可以同时进行，对用户来说，不用等待了，不用等待上一个操作就可以执行新的操作了。给用户的体验增强了。异步传输是面向字符的传输，它的单位是字符；而同步传输是面向比特的传输，它的单位是帧，它传输的时候要求接受方和发送方的时钟是保持一致的。其实 ajax 并不是什么新技术，而是一些技术的组装。

Ajax 给我们的网络带来了很大的好处，假如我们只是简单的提交一个表单，我们就没必要刷新整个页面，只需要局部表单提交，刷新局部就可以了，这将大大减少了网络流量。有优点的同时他也有缺点，缺点就是我们浏览器上的后退和前进按钮就失效了。假如我们页面有三个异步 ajax 操作，浏览器并不把他当作三个请求操作，后退的时候也不会后退三次回到原始页面

Ajax 的工作原理相当于在用户和服务器之间加了一个中间层,使用户操作与服务器响应异步化。并不是所有的用户请求都提交给服务器,像一些数据验证和数据处理等都交给 Ajax 引擎自己来做,只有确定需要从服务器读取新数据时再由 Ajax 引擎代为向服务器提交请求。

Ajax 中最重要的一个对象就是 XMLHttpRequest，这个对象最早是由微软在 IE 中以插件的形式提供的，但微软只是简单的提出这么一个对象，并没有真正的去实用它，后来其他浏览器也给出了这个对象，但他们都是提供的内置

对象，而不是简单的插件的形式了。所以在获得这个对象的时候就避免不了要用 if。。Else 判断了、判断是否为 IE 浏览器，我们使用判断一个对象是否存在来判断，这个对象是 IE 所特有的，他就是 active 控件的对象。通过 window.ActiveXObject，在 if 语句中写这么语句，因为在 javascript 中，如果不是 undefined 或者 false 他就是为 true，所以，只要这个 active 空间存在，if 条件就为真。也就是说就是 IE 浏览器。IE 获得 XMLHttpRequest 对象是一个固定形式：

```
xmlHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
```

这是 IE 特有的获得 XMLHttpRequest 对象的方式，其他浏览要想获得 XMLHttpRequest 对象直接 new 就可以，也就是 `xmlHttpRequest = new XMLHttpRequest()`；虽然 IE 和其他浏览器获得 XMLHttpRequest 对象的方式是不同的，但是 XMLHttpRequest 的使用方式是一样的。

好了，判断完浏览器之后，我们就开始准备向服务器发送请求了，准备发送请求我们用：`xmlHttpRequest.open("POST", "AjaxServlet", true)`；这里的三个参数我们有必要说一下，第一个参数是表示我们的请求是以什么形式发送，第二个参数是我们请求的地址，这里我们的地址是一个 **servlet**，第三个参数指明是否为以异步的形式发送请求，一般我们都会设置为 true。

准备好发送以后，我们要给他接收做一下准，ajax 接收数据是以一个回调函数的形式接收数据的。也就是说我们注册好这个回调函数后，当达到某一要求时他会自动调用这个回调函数，然后去执行回调函数的内容，注册回调函数：`xmlHttpRequest.onreadystatechange = ajaxCallback`；注意这个回调

函数 `ajaxCallback` 不要带括号。这个回调函数注册点我们从字面上也可以看出,他是在准备状态改变的时候调用这个函数。

一切准备好之后我们就要进行真正的发送请求了，发送请求是调用的 `xmlHttpRequest` 对象的 `send` 方法，这个方法里面当以 `post` 的形式发送请求，这里的方法的参数，就是我们请求的地址的参数，这个地址的参数是以键值对的形式传参的，如果是以 `get` 的形式请求时，我们可以把 `send` 方法的参数设置为 `null`。，假如我们是以 `get` 的请求方式，发送代码即为了：
`xmlHttpRequest.send(null);`，我们在来说说以 `post` 的请求方式请求时。
`xmlHttpRequest.send(null)`，`null` 可以传参数（即 `send（“v1="+v1）`）；
并且在真正发送之前（`xmlHttpRequest.send(null)`）之前必须设置

`xmlHttpRequest.setRequestHeader("content-type","application/x-www-form-urlencoded");`这一点一定要注意。

OK，请求发送出去了之后，我们下面来看一下我们怎么来接受请求返回的数据。根据 HTTP 协议我们应该知道，我们的一个请求应该分为四个步骤，也就是说一个请求有四个状态，他的状态即为 `xmlHttpRequest` 对象的 `readyState` 属性。我们来看一下这五个状态的具体内容：

从上边可以看出我们上边注册的回调函数将会被执行四次，但是我们其实就只在请求完成时执行我们回调函数的内容就 OK，所以在回调函数里面我们可以进行一下判断

```
if (xmlHttpRequest.readyState == 4) { //请求完成}。
```

虽然我们判断了请求是否完成，但我们不知道这个请求是否成功，在我们 http 协议中，请求成功的状态码是 200，所以我们如下判断一下状态码是否为 200 就可以了。

```
if (xmlHttpRequest.status == 200) {}
```

OK，以上差不多我们就把 ajax 的执行过程讲解了一遍，在看具体代码之前，我们来看一下 xmlHttpRequest 这个对象的属性：

下面我们就来看一下 ajax 整个执行流程的代码示例：我们以一个计算器的例子来实现：

首先我们来看一下表单代码：

[html] [view plaincopyprint?](#)

```
1. <input type="text" value="0" />
2. onclick=
3.      ();
4. >
5. <br>
6. <input type="button" value="+" name="add" id="add" />
7. <br>
8. <input type="button" value="=" name="calc" id="calc" />
9. <br>
10. <div id="result"></div>
```

然后我们来看一下执行的 ajax 的 javascript 代码：

[javascript] [view plaincopyprint?](#)

```
1. <script type="text/javascript">
2. var xmlHttpRequest = null; //声明一个空对象以接收 XMLHttpRequest 对象
3. function ajaxSubmit() {
4. if (window.ActiveXObject) // IE 浏览器
```

```

5.  {
6.  xmlHttpRequest = new ActiveXObject("Microsoft.XMLHTTP");
7.  } else if (window.XMLHttpRequest) //除 IE 外的其他浏览器实现
8.  {
9.  xmlHttpRequest = new XMLHttpRequest();
10.}
11.if (null != xmlHttpRequest) {
12.  var v1 = document.getElementById("value1ID").value;
13.var v2 = document.getElementById("value2ID").value;
14.  //当利用 get 方法访问服务器端时带参数的话，直接在"AjaxServlet"后面加参
    数，          下面 send 方法为参数 null 就可以，用 post 方法这必须在把参数加
    在 send 参数内，如下
15.xmlHttpRequest.open("POST", "AjaxServlet", true);
16.//关联好 ajax 的回调函数
17.xmlHttpRequest.onreadystatechange = ajaxCallback;
18.//真正向服务器端发送数据
19.// 使用 post 方式提交，必须要加上如下一行,get 方法就不必加此句
20.xmlHttpRequest.setRequestHeader("Content-Type",
21."application/x-www-form-urlencoded");
22.xmlHttpRequest.send("v1=" + v1 + "&v2=" + v2);
23.}
24.}
25.function ajaxCallback() { //ajax 一次请求会改变四次状态,所以我们在第四次(即一次
    请求结束)进行处理就 OK,
26.if (xmlHttpRequest.readyState == 4) { //请求成功
27.if (xmlHttpRequest.status == 200) {
28.var responseText = xmlHttpRequest.responseText;
29.document.getElementById("div1").innerHTML=responseText;
30.}
31.}
32.}
33.</script>

```

最后我们来看一下服务器端的执行：

Dopost 方法内

[java] view plaincopyprint?

```
1. String v1 = req.getParameter("v1");
2. String v2 = req.getParameter("v2");
3. System.out.println("v1=" + v1 + ", v2=" + v2);
4. String v3 = String.valueOf(Integer.valueOf(v1) + Integer.valueOf(v2));
5. PrintWriter out = resp.getWriter();
6. resp.setHeader("pragma", "no-cache");
7. resp.setHeader("cache-control", "no-cache");
8. out.println(v3);
9. out.flush();
```

下面我们来总结一下 **ajax** 的优势:

- 1、减轻服务器的负担。因为 **Ajax** 的根本理念是“按需取数据”，所以最大可能在减少了冗余请求和响影对服务器造成的负担；
- 2、无刷新更新页面，减少用户实际和心理等待时间；
- 3、也可以把以前的一些服务器负担的工作转嫁到客户端，利于客户端闲置的处理能力来处理，减轻服务器和带宽的负担，节约空间和带宽租用成本；
- 4、**Ajax** 使 **WEB** 中的界面与应用分离（也可以说是数据与呈现分离）；

（八十五）跟我学 jquery（一）爱之初体验 jquery

一、Jquery 简介

Jquery 是一个优秀的 JavaScript 框架。它是轻量级的 js 库(压缩后只有 21k)，它兼容 CSS3，还兼容各种浏览

器（IE 6.0+, FF 1.5+, Safari 2.0+, Opera 9.0+）。Jquery 应用到我们的项目中能够使程序员从设计和书写繁杂的 JS 应用中解脱出来，将关注点转向功能需求而非实现细节上，从而提高项目的开发速度。它有助于简

化 JavaScript 以及 Ajax 编程。它能让你在你的网页上简单的操作文档、处理事件、实现特效并为 Web 页面添加 Ajax 交互。

官方站点：<http://jquery.com/> 中文站点：<http://jquery.org.cn/>

二、jquery 的优点：

- 1、代码简练、语义易懂、学习快速、文档丰富。
- 2、jQuery 是一个轻量级的脚本，其代码非常小巧，最新版的 JavaScript 包只有 20K 左右。
- 3、jQuery 支持 CSS1-CSS3,以及基本的 XPath。
- 4、jQuery 是跨浏览器的，它支持的浏览器包括 IE 6.0+, FF 1.5+, Safari 2.0+, Opera 9.0+。
- 5、可以很容易的为 jQuery 扩展其他功能。
- 6、能将 JS 代码和 HTML 代码完全分离，便于代码和维护和修改。
- 7、插件丰富，除了 jQuery 本身带有的一些特效外，可以通过插件实现更多功能，如表单验证、tab 导航、拖放效果、表格排序、DataGrid，树形菜单、图像特效以及 ajax 上传等。

三、jquery 的基本特点和语法

在需要使用 JQuery 的页面中引入 JQuery 的 js 文件即可以使用 jquery 了。

例如：<script type="text/javascript" src="js/jquery.js"></script>引入之后便可在页面的任意地方使用 jQuery 提供的语法。

1、关于页面元素的引用

通过 jquery 的\$()引用元素包括通过 id、class、元素名以及元素的层级关系及 dom 或者 xpath 条件等方法，且返回的对象为 jquery 对象（集合对象），不能直接调用 dom 定义的方法。

2、jQuery 对象与 dom 对象的转换

只有 jquery 对象才能使用 jquery 定义的方法。注意 dom 对象和 jquery 对象是有区别的，调用方法时要注意操作的是 dom 对象还是 jquery 对象。

普通的 dom 对象一般可以通过\$()转换成 jquery 对象。

如：\$(document.getElementById("msg"))则为 jquery 对象，可以使用 jquery 的方法。

由于 jquery 对象本身是一个集合。所以如果 jquery 对象要转换为 dom 对象则必须取出其中的某一项，一般可通过索引取出。

如：\$("#msg")[0]，\$("div").eq(1)[0]，\$("div").get()[1]，\$("td")[5]这些都是 dom 对象，可以使用 dom 中的方法，但不能再使用 JQuery 的方法。

以下几种写法都是正确的：

```
$("#msg").html();  
$("#msg")[0].innerHTML;  
$("#msg").eq(0)[0].innerHTML;  
$("#msg").get(0).innerHTML;
```

3、如何获取 jQuery 集合的某一项

对于获取的元素集合，获取其中的某一项（通过索引指定）可以使用 `eq` 或 `get(n)` 方法或者索引号获取，**要注意，`eq` 返回的是 jquery 对象，而 `get(n)` 和索引返回的是 dom 元素对象。**对于 jquery 对象只能使用 jquery 的方法，而 dom 对象只能使用 dom 的方法，如要获取第三个<div>元素的内容。有如下两种方法：

```
$("#div").eq(2).html();           //调用 jquery 对象的方法
$("#div").get(2).innerHTML;       //调用 dom 的方法属性
```

4、同一函数实现 `set` 和 `get`

Jquery 中的很多方法都是如此，主要包括如下几个：

```
$("#msg").html(); //返回 id 为 msg 的元素节点的 html 内容。
$("#msg").html("<b>new content</b>");
//将"<b>new content</b>" 作为 html 串写入 id 为 msg 的元素节点内容中,页
面显示粗体的 new content

$("#msg").text(); //返回 id 为 msg 的元素节点的文本内容。
$("#msg").text("<b>new content</b>");
//将"<b>new content</b>" 作为普通文本串写入 id 为 msg 的元素节点内容中,
页面显示<b>new content</b>

$("#msg").height();           //返回 id 为 msg 的元素的高度
$("#msg").height("300");       //将 id 为 msg 的元素的高度设为 300

$("#msg").width();            //返回 id 为 msg 的元素的宽度
$("#msg").width("300");        //将 id 为 msg 的元素的宽度设为 300

$("#input").val("");           //返回表单输入框的 value 值
$("#input").val("test");       //将表单输入框的 value 值设为 test
```



```
$("#msg").click();    //触发 id 为 msg 的元素的单击事件
```

```
$("#msg").click(fn);    //为 id 为 msg 的元素单击事件添加函数
```

同样 blur,focus,select,submit 事件都可以有着两种调用方法

5、集合处理功能

对于 jquery 返回的集合内容无需我们自己循环遍历并对每个对象分别做处理，jquery 已经为我们提供的很方便的方法进行集合的处理。

包括两种形式：

```
$("#p").each(function(i){this.style.color=['#f00','#0f0','#00f'][i]})
```

//为索引分别为 0，1，2 的 p 元素分别设定不同的字体颜色。

```
$("#tr").each(function(i){this.style.backgroundColor=['#ccc','#fff'][i%2]})
```

//实现表格的隔行换色效果

介绍一下 **each** 方法：以每一个匹配的元素作为上下文来执行一个函数。意味着，每次执行传递进来的函数时，函数中的 **this** 关键字都指向一个不同的 DOM 元素（每次都是一个不同的匹配元素）。而且，在每次执行函数时，都会给函数传递一个表示作为执行环境的元素在匹配的元素集合中所处位置的数字值作为参数（从零开始的整型）。返回 **'false'** 将停止循环（就像在普通的循环中使用 **'break'**）。返回 **'true'** 跳至下一个循环（就像在普通的循环中使用 **'continue'**）。

另一种形式：

```
$("#p").click(function(){alert($(this).html())})
```

//为每个 p 元素增加了 click 事件，单击某个 p 元素则弹出其内容

6、扩展我们需要的功能

```
$.extend({  
    min: function(a, b){return a < b?a:b; },
```

```
max: function(a, b){return a > b?a:b; }  
}); //为 jquery 扩展了 min,max 两个方法
```

使用扩展的方法（通过“\$.方法名”调用）：

```
alert("a=10,b=20,max="+$.max(10,20)+" ,min="+$.min(10,20));
```

7、支持方法的连写

所谓连写，即可以对一个 jquery 对象连续调用各种不同的方法。

例如：

```
$("p").click(function(){alert($(this).html())})  
.mouseover(function(){alert('mouse over event')})  
.each(function(i){this.style.color=['#f00','#0f0','#00f'][i]});
```

8、操作元素的样式

主要包括以下几种方式：

```
$("#msg").css("background"); //返回元素的背景颜色  
$("#msg").css("background","#ccc") //设定元素背景为灰色  
$("#msg").height(300); $("#msg").width("200"); //设定宽高  
$("#msg").css({ color: "red", background: "blue" });//以名值对的形式设定样式  
式
```

```
$("#msg").addClass("select"); //为元素增加名称为 select 的 class  
$("#msg").removeClass("select"); //删除元素名称为 select 的 class  
$("#msg").toggleClass("select"); //如果存在（不存在）就删除（添加）名称为 select 的 class
```

9、完善的事件处理功能

Jquery 已经为我们提供了各种事件处理方法，我们无需在 html 元素上直接写事件，而可以直接为通过 jquery 获取的对象添加事件。

如:

```
$("#msg").click(function(){alert("good")}) //为元素添加了单击事件
```

```
$("#p").click(function(i){this.style.color=['#f00','#0f0','#00f'][i]})
```

//为三个不同的 p 元素单击事件分别设定不同的处理

jQuery 中几个自定义的事件:

(1) hover(fn1,fn2): 一个模仿悬停事件（鼠标移动到一个对象上面及移出这个对象）的方法。当鼠标移动到一个匹配的元素上面时，会触发指定的第一个函数。当鼠标移出这个元素时，会触发指定的第二个函数。

//当鼠标放在表格的某行上时将 class 置为 over，离开时置为 out。

```
$("#tr").hover(function(){
    $(this).addClass("over");
},
    function(){
        $(this).removeClass("over");
    });
```

(2) ready(fn):当 DOM 载入就绪可以查询及操纵时绑定一个要执行的函数。

```
$(document).ready(function(){alert("Load Success")})
```

//页面加载完毕提示“Load Success”，不同于 onload 事件，onload 需要页面内容加载完毕（图片等），而 ready 只要页面 html 代码下载完毕即触发。与\$(fn)等价

(3) toggle(evenFn,oddFn): 每次点击时切换要调用的函数。如果点击了一个匹配的元素，则触发指定的第一个函数，当再次点击同一元素时，则触发指定的第二个函数。随后的每次点击都重复对这两个函数的轮番调用。

//每次点击时轮换添加和删除名为 selected 的 class。

```
$("#p").toggle(function(){
    $(this).addClass("selected");
```

```
    },function(){  
        $(this).removeClass("selected");  
    });
```

(4) trigger(eventtype): 在每一个匹配的元素上触发某类事件。

例如:

```
$( "p" ).trigger( "click" ); //触发所有 p 元素的 click 事件
```

(5) bind(eventtype,fn), unbind(eventtype): 事件的绑定与反绑定

从每一个匹配的元素中（添加）删除绑定的事件。

例如:

```
$( "p" ).bind( "click", function(){alert( $(this).text() );} ); //为每个 p 元素添加
```

单击事件

```
$( "p" ).unbind(); //删除所有 p 元素上的所有事件
```

```
$( "p" ).unbind( "click" ) //删除所有 p 元素上的单击事件
```

10、几个实用特效功能

其中 toggle()和 slidetoggle()方法提供了状态切换功能。

如 toggle()方法包括了 hide()和 show()方法。

slideToggle()方法包括了 slideDown()和 slideUp 方法。

11、几个有用的 jQuery 方法

\$.browser.浏览器类型: 检测浏览器类型。有效参数:

safari, opera, msie, mozilla。如检测是否 ie: \$.browser.isie, 是 ie 浏览器则返回 true。

\$.each(obj, fn): 通用的迭代函数。可用于近似地迭代对象和数组(代替循环)。

如

```
$.each( [0,1,2], function(i, n){ alert( "Item #" + i + ": " + n ); });
```

等价于：

```
var tempArr=[0,1,2];
for(var i=0;i<tempArr.length;i++){
    alert("Item #"+i+": "+tempArr[i]);
}
```

也可以处理 json 数据， 如

```
$.each( { name: "John", lang: "JS" }, function(i, n){ alert( "Name: " + i + ", Value: " + n ); });
```

结果为：

Name:name, Value:John

Name:lang, Value:JS

\$.extend(target,prop1,propN): 用一个或多个其他对象来扩展一个对象，返回这个被扩展的对象。这是 **jquery** 实现的继承方式。

如：

```
$.extend(settings, options);
```

//合并 **settings** 和 **options**，并将合并结果返回 **settings** 中，相当于 **options**

继承 **setting** 并将继承结果保存在 **setting** 中。

```
var settings = $.extend({}, defaults, options);
```

//合并 **defaults** 和 **options**，并将合并结果返回到 **setting** 中而不覆盖 **default** 内容。

可以有多个参数（合并多项并返回）

\$.map(array, fn): 数组映射。把一个数组中的项目(处理转换后)保存到另一个新数组中，并返回生成的新数组。

如：

```
var tempArr=$.map( [0,1,2], function(i){ return i + 4; });
```

tempArr 内容为: [4,5,6]

```
var tempArr=$.map( [0,1,2], function(i){ return i > 0 ? i + 1 : null; });
tempArr 内容为: [2,3]
```

\$.merge(arr1,arr2):合并两个数组并删除其中重复的项目。

如: `$.merge([0,1,2], [2,3,4])` //返回[0,1,2,3,4]

\$.trim(str): 删除字符串两端的空白字符。

如: `$.trim(" hello, how are you? ")`; //返回"hello,how are you? "

12、解决自定义方法或其他类库与 jQuery 的冲突

很多时候我们自己定义了\$(id)方法来获取一个元素，或者其他的一些 js 类库如 **prototype** 也都定义了\$方法，如果同时把这些内容放在一起就会引起变量方法定义冲突，**Jquery** 对此专门提供了方法用于解决此问题。

使用 **jquery** 中的 `jQuery.noConflict();`方法即可把变量\$的控制权让渡给第一个实现它的那个库或之前自定义的\$方法。之后应用 **Jquery** 的时候只要将所有的\$换成jQuery即可，如原来引用对象方法`$("#msg")`改为`jQuery("#msg")`。

如:

```
jQuery.noConflict();
// 开始使用 jQuery
jQuery("div p").hide();
// 使用其他库的 $()
$("#content").style.display = 'none';
```

我们最后来看一下我们的测试代码示例:

[javascript] [view plaincopyprint?](#)

1. `<script type="text/javascript">`
2. `$(document).ready(function(){`
3. `//$("#di").html("helloworld"); //id 选择器`

```

4. //$(document.getElementById("di")).html("nihao");//dom 对象转换成 jquery 对象
5. //$("#di").get(0).innerHTML="caoshenghuan";//jquery 对象转换成 dom 对象
6. //$("#di").text("曹胜欢");
7. // alert($("#di").text());//同一函数的 set、get 方法
8.
9. //$("#div").click(function (){alert("你好")});
10. //$("#div").each(function(i){this.style.backgroundColor=['#f00','#0f0','#00f'][i]});
11. //$("#div").each(function(i){this.style.backgroundColor=['#ccc','#fff'][i%2]}) ;//jquery 集合处理功能
12.
13.// $.extend({
14.    //      min: function(a, b){return a < b?a:b; },
15.    //      max: function(a, b){return a > b?a:b; }
16.    //      }); //为 jquery 扩展了 min,max 两个方法
17.// alert($.min(100,200));
18.//$("#di").click(function(){alert("click")}).each(function(i){this.style.backgroundColor=['#f00','#0f0','#00f'][i]});//支持方法的连写
19.// alert($("#di").css("background-color"));
20.// $("#di").css("background-color","white");//操作元素的样式
21.//$("#div").hover(function(){
22.    //      alert("on");
23.    //      }, //模仿悬停事件
24.    //      function(){
25.    //      alert("out");
26.    //      });
27. })
28.</script>
29.</script>
30.</head>
31.<body>
32.<div id="di" style="background-color:#002 ; width:50px; height:100px;" ></div>
33.<div id="di1" style="background-color:#002 ; width:50px; height:100px;"></div>
34.<div id="di2" style="background-color:#002 ; width:50px; height:100px;"></div>
35.</body>
36.</html>

```


（八十六）跟我学 jquery（二）大话 jquery 选择器

本篇博客我将带大家来学习一下 jquery 的第一个比较重要的知识点，这个知识点对学习 jquery 的同学来说是必须掌握的，因为他是所有操作的基础，这个知识点就是 jquery 的对象选择器，我们利用 jquery 的操作都是基于对象上的，我们只有正确的选择好了我们要操作的对象，我们才能进行我们下一步的操作。jQuery 的选择器是什么方便的，我们几乎可以利用它获取页面上任意的一个或一组对象, 可以明显减轻开发人员的工作量。

什么是 jquery 选择器

在 Dom 编程中我们只能使用有限的函数根据 id 或者 TagName 获取 Dom 对象. 在 jQuery 中则完全不同, jQuery 提供了异常强大的选择器用来帮助我们获取页面上的对象, 并且将对象以 **jQuery 包装集** 的形式返回.

首先来看看什么是选择器:

```
//根据 ID 获取 jQuery 包装集  
var Object = $("#testDiv");
```

上例中使用了 ID 选择器, 选取 id 为 testDiv 的 Dom 对象并将它放入 jQuery 包装集, 最后以 jQuery 包装集的形式返回。"\$"符号在 jQuery 中代表对 jQuery 对象的引用, "jQuery"是核心对象, 其中包含下列方法:

jQuery(expression, context)

Returns: jQuery

这个函数接收一个 CSS 选择器的字符串，然后用这个字符串去匹配一组元素。

This function accepts a string containing a CSS selector which is then used to match a set of elements.

`jQuery(html, ownerDocument)`

Returns: jQuery

根据 HTML 原始字符串动态创建 Dom 元素.

Create DOM elements on-the-fly from the provided String of raw HTML.

`jQuery(elements)`

Returns: jQuery

将一个或多个 Dom 对象封装 jQuery 函数功能(即封装为 jQuery 包装集)

Wrap jQuery functionality around a single or multiple DOM Element(s).

`jQuery(callback)`

Returns: jQuery

`$(document).ready()`的简写方式

A shorthand for `$(document).ready()`.

上面摘自 jQuery 官方手册>Returns 的类型为 jQuery 即表示返回的是

jQuery 包装集.

根据选择器选取匹配的对象, 以 jQuery 包装集的形式返回. context 可以是

Dom 对象集合或 jQuery 包装集, 传入则表示要从 context 中选择匹配的对

象, 不传入则表示范围为文档对象(即页面全部对象).

上面这个方法就是我们选择器使用的核心方法.可以用"\$"代替 jQuery 让语法

更简介, 比如下面两句话的效果相同:

```
//根据 ID 获取 jQuery 包装集
```

```
var jQueryObject = $("#testDiv");
```

```
//$是 jQuery 对象的引用:
```

```
var jQueryObject = jQuery("#testDiv");
```

接下来让我们系统的学习 jQuery 选择器.

1. 基础选择器 Basics

名称	说明	举例
#id	根据元素 Id 选择	<code>\$("#divId")</code> 选择 ID 为 divId 的元素
element	根据元素的名称选择,	<code>\$("a")</code> 选择所有<a>元素
.class	根据元素的 css 类选择	<code>\$(".bgRed")</code> 选择所用 CSS 类为 bgRed 的元素
*	选择所有元素	<code>\$("*")</code> 选择页面所有元素
selector1, selector2, selectorN	可以将几个选择器用","分隔开 然后再拼成一个选择器字符串.会同时选中这几个选择器匹配的内容.	<code>\$("#divId, a, .bgRed")</code>

2.层次选择器 Hierarchy

名称	说明	举例
ancestor descendant	使用"form input"的形式 选中 form 中的所有 input 元素.即 ancestor(祖先) 为 from, descendant(子 孙)为 input.	<code>\$(".bgRed div")</code> 选择 选中 form 中的所有 input 元素.即 ancestor(祖先) 为 from, descendant(子 孙)为 input.
parent > child	选择 parent 的直接子节	<code>\$(".myList>li")</code> 选择

点 child. child 必须包含 CSS 类为 myList 元素
在 parent 中并且父类是 中的直接子节点对
parent 元素. 象.

prev + next

prev 和 next 是两个同级 \$("#hibiscus+img")选
别的元素. 选中在 prev 元 在 id 为 hibiscus 元素后
素后面的 next 元素. 面的 img 对象.

prev ~ siblings

选择 prev 后面的根据 \$("#someDiv~[title]")选
siblings 过滤的元素 择 id 为 someDiv 的对
注:siblings 是过滤器 象后面所有带有 title 属
性的元素

3.基本过滤器 Basic Filters

名称	说明	举例
:first	匹配找到的第 一个元素	查找表格的第一行:\$("#tr:first")
:last	匹配找到的最 后一个元素	查找表格的最后一行:\$("#tr:last")
:not(select or)	去除所有与给 定选择器匹配 的元素	查找所有未选中的 input 元 素: \$("#input:not(:checked)")
:even	匹配所有索引	查找表格的 1、3、5...行:\$("#tr:even")

值为偶数的元

素，从 0 开始

计数

:odd

匹配所有索引 查找表格的 2、4、6 行:\$(**"tr:odd"**)

值为奇数的元

素，从 0 开始

计数

:eq(index)

匹配一个给定 查找第二行:\$(**"tr:eq(1)"**)

索引值的元

素

注:index

从 0 开始计数

:gt(index)

匹配所有大于 查找第二第三行，即索引值是 1 和 2，也就是

给定索引值的 比 0 大:\$(**"tr:gt(0)"**)

元素

注:index

从 0 开始计数

:lt(index)

选择结果集中 查找第一第二行，即索引值是 0 和 1，也就是

索引小 比 2 小:\$(**"tr:lt(2)"**)

于 N 的 elem

ents

注:index

从 0 开始计数

:header

选择所有 给页面内所有标题加上背景

h1,h2,h3 一类 色: \$(" :header").css("background", "#EEE");

的 header 标

签.

:animated

匹配所有正在 只有对不在执行动画效果的元素执行一个动

执行动画效果 画特效:

的元素

```
$("#run").click(function(){  
    $("div:not(:animated)").animate({ left: "+=20" }, 1000);  
});
```

4. 内容过滤器 **Content Filters**

名称

说明

举例

:contains(text)

匹配包 查找所有包含 "John" 的 div 元

含给定 素: \$("div:contains('John')")

文本的

元素

:empty

匹配所 查找所有不包含子元素或者文本的空元

有不包 素: \$("td:empty")

含子元

素或者

文本的

空元素

:has(selector) 匹配含 给所有包含 p 元素的 div 元素添加一
 有选择 个 text 类: \$("div:has(p)").addClass("test");
 器所匹
 配的元
 素的元
 素

:parent 匹配含 查找所有含有子元素或者文本的 td 元
 有子元 素:\$("td:parent")
 素或者
 文本的
 元素

5.可见性过滤器 Visibility Filters

名称	说明	举例
:hidden	匹配所有的不可见元素 注:在 1.3.2 版本中, hidden 匹配自身或者 的 tr 元 父类在文档中不占用空间的元素.如果使 素:\$("tr:hidden") 用 CSS visibility 属性让其不显示但是占 位,则不输入 hidden.	查找所有不可见
:visible	匹配所有的可见元素	查找所有可见的 tr 元 素:\$("tr:visible")

6.属性过滤器 Attribute Filters

名称

说 举例

明

[attribute]

匹 查找所有含有 id 属性的 div 元素:

配 `$("div[id]")`

包

含

给

定

属

性

的

元

素

[attribute=value]

匹 查找所有 name 属性

配 是 newsletter 的 input 元素:

给 `$("input[name='newsletter']").attr("checked", true);`

定

的

属

性

是

某个特定值的元素

`[attribute!=value]`

匹 查找所有 name 属性不

配 是 newsletter 的 input 元素:

给 `$("input[name!='newsletter']").attr("checked", true);`

定的属性是不包含某个特定

[attribute^=value]

定
值
的
元
素
匹 配
给
定
的
属
性
是
以
某
些
值
开
始
的
元
素

\$("input[name^='news']")

[attribute\$=value]

匹 查找所有 name 以 'letter' 结尾

配 的 input 元素:

给 \$("input[name\$='letter']")

定

的

属

性

是

以

某

些

值

结

尾

的

元

素

[attribute*=value]

匹 查找所有 name 包

配 含 'man' 的 input 元素:

给 \$("input[name*='man']")

定

的

属性是以包含某些值的元素的

`[attributeFilter1][attributeFilter2][attributeFilterN]`

复 找到所有含有 id 属性，并且它
合 的 name 属性是以 man 结尾的：
属 `$("input[id][name$='man']")`

性
选
择
器
,
需
要

同时满足多个条件时使用。

7.子元素过滤器 Child Filters

名称	说明	举例
<code>:nth-child(index/even/odd/equation)</code>	匹配其父元素下的第 N 个子或奇偶元素	在每个 ul 查找第 2 个 li: <code>\$("ul li:nth-child(2)")</code>
	<code>'eq(index)'</code> 只匹配一个元素，而这个将为每一个父元素匹	

配子元

素。`:nth-child` 从

1 开始的，

而`:eq()`是从 0 算

起的！

可以使用：

```
nth-child(even)
:nth-child(odd)
:nth-child(3n)
:nth-child(2)
:nth-child(3n+1)
)
:nth-child(3n+2)
)
```

`:first-child`

匹配第一个子 在每个 `ul` 中查找第

元素 一个 `li`：

`:first` 只匹配一 `$("ul li:first-child")`

个元素，而此选

择符将为每个

父元素匹配一

个子元素

`:last-child`

匹配最后一个 在每个 `ul` 中查找最

子元素 后一个 `li`：

`:last` 只匹配一 `$("ul li:last-child")`

个元素，而此选

择符将为每个

父元素匹配一

个子元素

:only-child

如果某个元素 在 ul 中查找是唯一

是父元素中唯 子元素的 li:

一的子元素, 那 `$("ul li:only-child")`

将会被匹配

如果父元素中

含有其他元素,

那将不会被匹

配。

8. 表单选择器 Forms

名称	说明	解释
:input	匹配所有 input, textarea, select 和 button 元素	查找所有的 input 元素: <code>\$(":input")</code>
:text	匹配所有的文本框	查找所有文本框: <code>\$(":text")</code>
:password	匹配所有密码框	查找所有密码框: <code>\$(":password")</code>
:radio	匹配所有单选按钮	查找所有单选按钮
:checkbox	匹配所有复选框	查找所有复选框:

:submit	匹配所有提交按钮	<code>\$(":checkbox")</code> 查找所有提交按钮:
:image	匹配所有图像域	<code>\$(":submit")</code> 匹配所有图像域:
:reset	匹配所有重置按钮	<code>\$(":image")</code> 查找所有重置按钮:
:button	匹配所有按钮	<code>\$(":reset")</code> 查找所有按钮:
:file	匹配所有文件域	<code>\$(":button")</code> 查找所有文件域:
		<code>\$(":file")</code>

这里要注意一点的是在表单选择器前一定要注意空格问题

[javascript] [view plaincopyprint?](#)

```

1. <script type="text/javascript">
2. $(function()
3. {
4. alert($('.test :hidden').length); //选择 class 为 test 的元素当中的隐藏子元素
5. alert($('.test:visible').length); //选择隐藏的 class 为 test 的元素
6. });
7. </script>
8. </head>
9. <body>
10. <div class="test">
11.   <div style="display:none">aaaa</div>
12.   <div style="display:none">bbbb</div>
13.   <div style="display:none">cccc</div>
14.   <div class="test" style="display:none">dddd</div>
15. </div>

```



```
16. <div class="test" style="display:none">eeee</div>
17. </body>
18. </html>
```

9. 表单过滤器 Form Filters

名称	说明	解释
:enabled	匹配所有可用元素	查找所有可用的 input 元素: \$("input:enabled")
:disabled	匹配所有不可用元素	查找所有不可用的 input 元素: \$("input:disabled")
:checked	匹配所有选中的被选中元素(复选框、单选框等, 不包括 select 中的 option)	查找所有选中的复选框元素: \$("input:checked")
:selected	匹配所有选中的 option 元素	查找所有选中的选项元素: \$("select option:selected")

以上基本上可以说是包含了大部分的 jquery 选择器的内容了, 其实这么多的内容一下子要全记住那也不太现实, 关键是要理解, 用的时候能查到, 但最好还是要记住, 一个人对 jquery 选择器的掌握, 很大一部分可以反映出他对 jquery 的掌握, 所以掌握好 jquery 选择器对以后的学习也是比较重要的一个环节

如果对本文很多函数不知道什么用途, 推荐阅读:

跟我学 **jquery**（三）**jquery** 动态创建元素和常用函数示例

最后给大家推荐几个文档：

jQuery 官方 API: <http://docs.jquery.com/>

中文在线 API: <http://jquery.org.cn/visual/cn/index.xml>

中文 jQuery API 下

载: <http://download.csdn.net/detail/csh624366188/4378360>

（八十七）跟我学 jquery（三）jquery 动态创建元素和常用函数示例

在上面两篇博客中列举了太多的 API 相信大家看着眼晕. 不过这些基础还必须要讲, 基础要扎实. 其实对于这些列表大家可以跳过, 等以后用到时再回头看或者查询官方的 API 说明. 在本博客中就给大家讲解一下这些头晕的 API 主要讲解动态创建元素操作 jQuery 包装集的各个函数

一. 动态创建元素

（这块转至网络，具体地址不详，以前找到的资料）

1. 错误的编程方法

我们经常使用 javascript 动态的创建元素, 有很多程序员通过直接更改某一个容器的 HTML 内容. 比如:

[html] [view plaincopyprint?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
   w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns=
   >
3. <head>
4.   <title>动态创建对象</title>
5. </head>
6. <body>
7.   <div id=
   >测试图层</div>
8. <script type=
   >
9.   document.getElementById("testDiv").innerHTML = "<div style=\\
   :solid 1p
   x #FF0000\\>动态创建的 div</div>";
10. </script>
11. </body>
12. </html>
```

上面的示例中我通过修改 testDiv 的内容,在页面上动态的添加了一个 div 元素. 但是请牢记,这是错误的做法!

错误的原因:

(1) 在页面加载时改变了页面的结构. 在 IE6 中如果网络变慢或者页面内容太大就会出现"终止操作"的错误. 也就是说"永远不要在页面加载时改变页面的 Dom 模型".

(2) 使用修改 HTML 内容添加元素, 不符合 Dom 标准. 在实际工作中也碰到过使用这种方法修改内容后, 某些浏览器中并不能立刻显示添加的元素, 因为不同浏览器的显示引擎是不同的. 但是如果我们使用 Dom 的 CreateElement 创建对象, 在所有的浏览器中几乎都可以. 但是在 jQuery 中如果传入的而是一个完整的 HTML 字符串, 内部也是使用 innerHTML. 所以也不是完全否定 innerHTML 函数的使用.

所以从现在开始请摒弃这种旧知识, 使用下面介绍的正确方法编程.

2.创建新的元素

下面介绍两种正确的创建元素的方式.

(1)使用 HTML DOM 创建元素

什么是 DOM?

通过 JavaScript, 您可以重构整个 HTML 文档。您可以添加、移除、改变或重排页面上的项目。

要改变页面的某个东西, JavaScript 就需要对 HTML 文档中所有元素进行访问的入口。这个入口, 连同对 HTML 元素进行添加、移动、改变或移除的方法和属性, 都是通过文档对象模型来获得的 (DOM)。

在 1998 年, W3C 发布了第一级的 DOM 规范。这个规范允许访问和操作 HTML 页面中的每一个单独的元素。

所有的浏览器都执行了这个标准, 因此, DOM 的兼容性问题也几乎难觅踪影了。

DOM 可被 JavaScript 用来读取、改变 HTML、XHTML 以及 XML 文档。

DOM 被分为不同的部分 (核心、XML 及 HTML) 和级别 (DOM Level 1/2/3):

Core DOM

定义了一套标准的针对任何结构化文档的对象

XML DOM

定义了一套标准的针对 XML 文档的对象

HTML DOM

定义了一套标准的针对 HTML 文档的对象。

关于使用 HTML DOM 创建元素本文不做详细介绍, 下面举一个简单的例子:

[html] [view plaincopyprint?](#)

```
1. //使用 Dom 标准创建元素
2. var select = document.createElement("select");
3. select.options[0] = new Option("加载项 1", "value1");
4. select.options[1] = new Option("加载项 2", "value2");
5. select.size = 1;
6. var object = document.appendChild(select);
```

通过使用 `document.createElement` 方法我们可以创建 Dom 元素, 然后通过 `appendChild` 方法为添加到指定对象上.

(2) 使用 jQuery 函数创建元素

在 jQuery 中创建对象更加简单, 比如创建一个 Div 元素:

```
$("<div style=\"border:solid 1px #FF0000\">动态创建的 div</div>")
```

我们主要使用 jQuery 核心类库中的一个方法:

`jQuery(html, ownerDocument)`

Returns: jQuery

根据 HTML 原始字符串动态创建 Dom 元素.

其中 `html` 参数是一个 HTML 字符串, 在 jQuery1.3.2 中对此函数做了改进:

当 HTML 字符串是没有属性的元素是, 内部使用 `document.createElement` 创建元素, 比如:

//jQuery 内部使用 `document.createElement` 创建元素:

```
$("<div/>").css("border","solid 1px #FF0000").html("动态创建的  
div").appendTo(testDiv);
```

否则使用 `innerHTML` 方法创建元素:

//jQuery 内部使用 `innerHTML` 创建元素:

```
$("<div style=\"border:solid 1px #FF0000\">动态创建的  
div</div>").appendTo(testDiv);
```

3.将元素添加到对象上

我们可以使用上面两种方式创建一个而元素, 但是上面已经提到一定不要在页面加载时就改变页面的 DOM 结构, 比如添加一个元素. 正确的做法是在页面加载完毕后添加或删除元素.

传统上, 使用 `window.onload` 完成上述目的:

//DOM 加载完毕后添加元素

//传统方法

```
window.onload = function() { testDiv.innerHTML = "<div style=\"border:solid 1px #FF0000\">动态创建的 div</div>"; }
```

虽然能够在 DOM 完整加载后, 在添加新的元素, 但是不幸的是浏览器执行 `window.onload` 函数不仅仅是在构建完 DOM 树之后, 也是在所有图像和其他外部资源完整的加载并且在浏览器窗口显示完毕之后. 所以如果某个图片或者其他资源加载很长时间, 访问者就会看到一个不完整的页面, 甚至在图片加载之前就执行了需要依赖动态添加的元素的脚本而导致脚本错误.

解决办法就是等 DOM 被解析后, 在图像和外部资源加载之前执行我们的函数. 在 jQuery 中让这一实现变得可行:

[javascript] [view plaincopyprint?](#)

1. //jQuery 使用动态创建的`$(document).ready(function)`方法
2. `$(document).ready(`
3. `function() { testDiv.innerHTML = "<div style=\"border:solid 1px #FF0000\">使用动态创建的$(document).ready(function)方法</div>"; }`
4. `);`

或者使用简便语法:

[javascript] [view plaincopyprint?](#)

1. //jQuery 使用\$(function)方法
2. \$(
3. function() { testDiv.innerHTML += "<div style=\"border:solid 1px #FF0000\">
 使用\$(function)方法</div>"; }
4.);

使用\$()将我们的函数包装起来即可. 而且可以在一个页面绑定多个函数, 如果使用传统的 window.onload 则只能调用一个函数.

所以请大家将修改 DOM 的函数使用此方法调用. 另外还要注意

document.createElement 和 innerHTML 的区别. 如果可以请尽量使用

document.createElement 和\$("<div/>")的形式创建对象.

二. Query 包装集的各个函数

1.文档加载完成执行函数

```
$(document).ready(function(){  
alert("开始了");  
});
```

2.添加/删除 CSS 类

```
$("#some-id").addClass("NewClassName");  
$("#some-id").removeClass("ClassNameToBeRemoved");
```

3.选择符 : 利用了 CSS 和 Xpath (XML Path Language) 选择符的能力, 以及 jQuery 独有的选择符

3.1 常用的:

1. 根据标签名: `$('p')` 选择文档中的所有段落
2. 根据 ID: `$("#some-id")`
3. 类: `$('.some-class')`

3.2 使用 CSS 选择符:

`$("#some-id > li")` 选择特定 id 下的所有子 li 元素

`$("#some-id li:not(.horizontal)")` 伪类选择, 特定 id 下所有没有 .horizontal 类的 li 元素

3.3 使用 XPath 选择符:

属性选择: `$(a[@title])` 选择所有带 title 属性的链接

`$(div[ol])` 选择包含一个 ol 元素的所有 div 元素

`$(a[@href^="mailto:%22"])` 选择所有带 href 属性[@href]且该属性值以 mailto 开头 ^= "mailto" (^标识字符串开始, \$标识字符串结尾, *表示字符串中任意位置)

`$(a[@href$=".pdf"])` 选择带有 href 属性且该属性值以 .pdf 结尾的所有链接

`$(a[@href*="mysite.com"])` 选择 mysite.com 出现在 href 任意位置(包含 mysite.com)的所有链接

3.4 JQuery 自定义选择符 (过滤器, 从已选择的结果集中过滤出符合某一条

件的所有元素)，与 CSS 的伪类选择符相似，使用“:”开头

1. `$('#div.horizontal:eq(1)')` 选择带有类 `horizontal` 的 `div` 集合中的第 2 个项

`$('#div:nth-child(1)')` 选择作为其父元素第 1 个子元素的所有 `div`

2. 自定义选择符 `:odd` 和 `:even` 选择奇偶行

`$('#tr:odd').addClass('odd');` //过滤选择结果集中的奇数元素

`$('#tr:even').addClass('even');` //过滤选择结果集中的偶数元素

3. 自定义选择符 `:contains()`

`$('#td:contains("Henry")')` 选择包含 `Henry` 字符串的所有表格

3.5 JQuery 选择函数

1. `$('#some-id').parent()` 选择特定元素的父元素

2. `$('#some-id').next()` 选择特定元素最近的下一个同级元素

3. `$('#some-id').siblings()` 选择特定元素的所有同级元素

4. `$('#some-id').find('.some-class')` 选择特定元素下所有包含特定类的元素

5. `$('#some-id').find('td').not(':contains("Henry")')` 选择特定元素下表格内容不包含 `Henry` 的所有元素

5. `$('#some-id').find('td').not(':contains("Henry")').end().end()` 表示回到最后一次 `.find()` 的元素处

3.6 访问 DOM 元素，使用 `get()` 方法从选择后的 JQuery 对象中获得，去掉 JQuery 的包装

`var myTag = $('#some-id').get(0).tagName;`

```
var myTag = $('#some-id')[0].tagName; //与上面的等效
```

4.事件（都是给某一元素绑定事件）

4.1 绑定事件

```
$("#some-id").bind("click", function(){ })
```

```
$("#some-id").unbind("click", bindedFunctionName); //移除已绑定的事件，
```

前提是绑定的函数有名称（不是匿名函数）

```
$("#some-id").click(function(){ })
```

4.2 复合函数绑定事件

```
$("#some-id").toggle(function(){ },function(){ }); //交替执行
```

```
$("#some-id").toggleClass("hidden"); // 添加/删除类交替进行
```

```
$("#some-id").hover(function(){ }, function(){ }); //鼠标进入元素时执行第一个函数，离开元素时执行第二个函数
```

```
$("#some-id").one("click", functionName); //只需触发一次，随后便立即解除绑定
```

4.3 模仿用户触发某一事件

```
$("#some-id").trigger("click"); //触发特定元素的 click 事件
```

5.为元素添加效果

5.1 读取或设置 CSS 样式属性

`$("#some-id").css("property")` //读取样式值

`$('#some-id').css('property', 'value')` //设置一个样式值

`$('#some-id').css({property1: 'value1', property2: 'value2'})` //设置多个样式属性

5.2 改变字体大小

```
$(document).ready({  
  $('#button-id').click(function(){  
    var currentSize = $('#text-id').css('fontSize'); //获取字体大小，如 30px  
    var num = parseFloat(currentSize, 10); //将值转为浮点数，.parseFloat(,)
```

为 javascript 内置函数，这里是转为 10 进制的浮点数

```
    var unit = currentSize.slice(-2); //获取单位名称，如 px，.slice()是 javascript  
    内置函数，获取字符串指从定位置开始的子字符串，-2 表示倒数两个字符  
    num *= 1.5;
```

```
    $('#text-id').css('fontSize', num + unit); //设置字体大小样式
```

```
  });
```

```
});
```

5.3 隐藏和显示

`$('#some-id').show();` //无效果，会自动记录原来的 `display` 属性值（如：
`block`, `inline`），再回复 `display` 值

`$('#some-id').hide();` //无效果，等效于：

`$('#some-id').css('display', 'none');` 隐藏元素，不保留物理位置

大小、透明度逐渐变化的显示或隐藏

`$('#some-id').show('slow');` //指定显示速度，在指定时间内元素的高、宽、不透明度逐渐增加到属性值，有：`slow` 是 0.6 秒，`normal` 是 0.4 秒，`fast` 是 0.2 秒，或者直接填入毫秒数

`$('#some-id').hide(800);` //与`.show()`指定速度显示一样，指定时间内高、宽、不透明度逐渐减小到 0

淡入淡出

`$('#some-id').fadeIn('slow');` //指定时间内不透明度属性值由 0 增加到 1

`$('#some-id').fadeOut('slow');` //指定时间内不透明度值由 1 减小到 0

5.4 构建具有动画效果的 show

主要调用`.animate()`方法，其有 4 个参数：包含样式属性及值的映射；可选的速度参数；可选的缓动类型；可选的回调函数；

1. 并发显示多个效果

`$('#some-id').animate({height: 'show', width: 'show', opacity: 'show'}, 'slow', function(){ alert('动画显示元素');});`

`$('#div .button').animate({left:600, height:44}, 'slow');` //水平位置从 0 移动到 600，高度由 0 增加到 44

2. 排队显示多个效果，级联多个`.animate()`，一个效果显示完了再显示下一个效果

`$('#some-id').animate({left:600}, 'slow').animate({height: 44}, 'slow');`

6. DOM 操作

6.1 属性操作

`$('#some-id').attr('property');` //获取属性

`$('#some-id').attr('property','value');` //设置属性

`$('#some-id').attr({'property1': 'value1', 'property2': 'value2'});` //设置多个属性

修改一个段落中所有链接，并给每个链接附上新的 id 号

```
$('#div p .content a').each(function(index){
$(this).attr({
'rel': 'external',
'id': 'link_' + index
});
});
```

// ***** JQuery 的.each()类似一个迭代器，给其传递的参数 index 类似一个计数器 *****

6.2 插入新元素

1.将元素插入到其他元素前面：`.insertBefore()`和`.before()`

2.将元素插入到其他元素后面：`.insertAfter()`和`after()`

3.将元素插入到其他元素内部或后面（相当于追加一个元素）：`appendTo()`和`append()`

4.将元素插入到其他元素内部或前面（相当于追加一个元素）：`prependTo()`
和 `prepend()`

6.3 包装元素，将元素包装到其他元素中 `.wrap()`;

`$('#some-id').wrap('');` // 将某一特定元素包装到 `li` 中，即在特定元素
外围添加一个包围元素

6.4 复制元素 `.clone()`

1.`$('#some-id').clone().appendTo($('body'));`

2.复制深度，当传入 `false` 参数时，只复制匹配上的元素，其内部的其他元素不复制

`$('#some-id').clone(false)`

注意：`.clone()`方法不会复制元素中的事件

6.5 移除匹配元素中的元素，类似清空元素

`$('#some-id').empty();`

6.6 从 DOM 中移除匹配的元素及其后代元素

`$('#some-id').remove();`

分享到：

（八十八）跟我学 jquery（四）JQuery 框架操作元素的属性与样式

在前面几篇博客中，我们初步了解了一下 jquery 的好处，基本语法，还有一些基本函数，这是学习 jquery 的基础，在这篇博客中，我们一起来学习一下 JQuery 框架操作元素的属性与样式，在 web 开发中，修改页面元素的属性和样式是我们需要常用的功能。所以掌握好这个知识点，对于我们在 web 开发中也是一个非常有利的利剑。好，下面我们就具体来看一下：

首先来看一下两个概念的区别：

元素属性和 Dom 属性

对于下面这样一个标签元素：

```
</img>
```

我们通常将 id,src,alt,class 称为属性,也即元素属性.但是,当浏览器对标签元素进行解析时,会将元素解析为 Dom 对象,相应的,元素属性也就解析为 Dom 属性.元素属性和 Dom 属性只是在我们对其进行不同解析时的不同称呼.

注意：

1.元素被解析成 Dom 时,元素属性和 Dom 属性并不一定是原来的名称.

例如,img 的 class 属性,在表现为元素属性时是 class;在表现为 Dom 属性时,属性名为 className

2.在 JavaScript 中,我们可以直接获取或设置 Dom 属性

JQuery 操作 DOM 属性

因为 JQuery 都是针对包装集进行操作，所以在 JQuery 中没有包装操作“DOM 属性”的函数。但是 JQuery 提供了 each()函数用于遍历 JQuery 包装

集，其中的 **this** 指针指向当前的 DOM 对象，我们可以应用这一点配合原生 javascript 来操作元素的 DOM 属性：

[javascript] [view plaincopyprint?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2. http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3. <html xmlns="http://www.w3.org/1999/xhtml">
4. <head>
5. <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6. <title>无标题文档</title>
7. <script src="jquery-1.4.js" type="text/javascript"></script>
8. <script type="text/javascript">
9. $(document).ready(function(){
10.     $.each($('img'),function(index) {
11.         alert("index:" + index + ", id:" + this.id + ", alt:" + this.alt);
12.         this.alt = "这里改变了！";
13.         alert("index:" + index + ", id:" + this.id + ", alt:" + this.alt);
14.     });
15. })
16. </script>
17. </head>
18. <body>
19. 
20. 
21. </body>
22. </html>
```

\$.each()函数的解析：上例代码中的`$.each()`中第一个参数为数组或者对象，第二个是回调函数，这个回调函数可以存在两个参数，第一个参数是数组或者对象的索引，第二个参数是数组或者对象的值。

jQuery 操作元素属性

JQuery 提供了属性 `attr()` 包装集函数, 能够同时操作包装集中所有元素的属性, 我们依旧通过实例来了解如何使用 JQuery 操作元素属性:

[javascript] [view plaincopyprint?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns="http://www.w3.org/1999/xhtml">
3. <head>
4. <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5. <title>无标题文档</title>
6. <script src="jquery-1.4.js" type="text/javascript"></script>
7. <script type="text/javascript">
8. $(document).ready(function(){
9.     alert($('#phpfuns1').attr('alt'));//获取 id 为 phpfuns1 的 alt 属性
10.    $('#phpfuns1').attr('alt','已经修改了!');//设置选定元素的单个属性值
11.    $('#phpfuns2').attr({'alt':'修改 alt 属性','name':'hello'})//设置选定元素的多个属性值
12.    $('#phpfuns3').attr('title',function(){//将回调函数计算出的值赋给元素属性
13.        return this.src;
14.    })
15.    $('#phpfuns1').removeAttr('alt');//移除元素的属性值
16.})
17.</script>
18.</head>
19.<body>
20.
21.
22.
23.</body>
24.</html>
```

上面的实例中包含了 **attr()** 包装集函数的各种用法。打开 **Firebug**，可以看到代码运行的结果。

在 **jQuery** 中,提供了 **attr** 函数来操作元素属性,具体如下:

函数名	说明	例子
	取得第一个匹配	
attr(name)	匹配元素的属性值	<code>\$("#input").attr("value")</code>
	将	
attr(property)	一	<code>\$("#input").attr({ value: "txt", title: "text"});</code>

个
"
名
/
值
"
形
式
的
对
象
设
置
为
所
有
匹
配
元
素
的
属
性

为所有匹配的元素的

attr(key,value)

素 `$("input").attr("value","txt");`

设置一个属性值为所有

attr(key,fn)

匹 `$("input").attr("title", function() { return this.value });`

配的元素

素
设
置
一
个
计
算
的
属
性
值
从
所
有
匹
配

removeAttr(name)

的 `$("input").removeAttr("value");`

元
素
中
删
除

一个属性

注意:

- 1.如果要设置对象的 **class** 属性时,必须使用 **className** 作为属性名.
- 2.我们可以使用 **removeAttr** 删除元素属性,但其对应的 **Dom** 属性是不会被删除掉的,只是被改变其值而已

实际上,jQuery 提供了更简单的方法来访问 **value**,**innerHTML**,具体函数如下:

函数名	说明	例子
val()	获取第一个匹配元素的 value 值	<code>\$("#txt1").val()</code>
val(val)	为匹配的元素设置 value 值	<code>\$("#txt1").val("txt1")</code>
html()	获取第一个匹配元素的 html 内容	<code>\$("#dv1").html()</code>
html(val)	设置每一个匹配的元素的 html 内容	<code>\$("#dv1").html("this is a div")</code>
text()	取得所有匹配文本节点的内容,并将其连接起来	<code>\$("div").text()</code>
text(val)	将所有匹配元素的置为 val	<code>\$("div").text("divs")</code>

JQuery 操作 CSS 样式

我们可以通过修改元素 CSS 类或者直接修改元素的样式来操作 CSS 样式。

1, JQuery 修改 CSS 类

一个元素可以应用多个 css 类，但是不幸的是在 DOM 属性中是用一个以空格分割的字符串存储的，而不是数组。所以如果在原始 javascript 时代我们想对元素添加或者删除多个属性时，都要自己操作字符串，而 JQuery 改变了这一切。

下面进入实例，看一下 JQuery 是如何操作 CSS 类的：

[javascript] [view plaincopyprint?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns="http://www.w3.org/1999/xhtml">
3. <head>
4. <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5. <title>无标题文档</title>
6. <script src="jquery-1.4.js" type="text/javascript"></script>
7. <script type="text/javascript">
8. $(document).ready(function(){
9.     $('#phpfuns1').addClass('test');//添加类名
10.    $('#phpfuns2').removeClass('test2');//删除类名，保留 class 属性
11.    if($('#phpfuns1').hasClass('test')){//判断包装集中是否至少有一个元素应用了指定的 CSS 类
12.        alert('111111111');
13.    }
14.})
15.</script>
16.</head>
17.<body>
18.
```

```
19.
20.</body>
21.</html>
```

使用上面的方法，我们可以将元素的 CSS 类像集合一样修改，再也不必手工解析字符串。

注意：`addClass(class)` 和 `removeClass([classes])` 的参数可以一次传入多个 css 类，用空格分割，比如：

```
$("#btnAdd").bind("click", function(event) { $("#p").addClass("colorRed borderBlue"); });
```

`removeClass` 方法的参数可选，如果不传入参数则移除全部 CSS 类：

```
$("#p").removeClass()
```

2, JQuery 修改 CSS 样式

同样当我们想要修改元素的具体某一个 CSS 样式，即修改元素属性"style"时，JQuery 也提供了相应的方法，下面我们通过实例来看一下如何使用 JQuery 修改 CSS 样式：

[javascript] [view plaincopyprint?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns="http://www.w3.org/1999/xhtml">
3. <head>
4. <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5. <title>无标题文档</title>
6. <script src="jquery-1.4.js" type="text/javascript"></script>
7. <script type="text/javascript">
8. $(document).ready(function(){
```

```

9.    $('p').css('color','red');//设置单一的 css 属性
10.   $('div').css({'width':'100px','background-color':'green','height':'20px'});//设置复合
      的 css 属性
11.   alert($('p').css('color'));//获取第一个匹配元素的样式属性
12.})
13.</script>
14.</head>
15.<body>
16.<p>这是 phpfun.com 的一个 JQuery 测试</p>
17.<div></div>
18.</body>
19.</html>

```

总结修改 css 样式方法如下：

1.修改 CSS 类

函数名	说明	例子
	为每个 匹配的 元素添 加指定 的类名	
addClass(classes)	判断匹 配元素 集合中 是否至 少有一	<code>\$("#input").addClass("colorRed borderBlack");</code>
hasClass(class)		<code>alert(\$("#input").hasClass("borderBlack"));</code>

个包含
了指定
的 **CSS**
类,如果
有一个
含有指
定 **CSS**
类,则返
回 **true**

从匹配
元素中

`removeClass([classes])` 移除所有或指

定的
CSS 类

如果存
在(不

`toggleClass(classes)` 存在)
就删除

(添加)
指定类

```
$("#input").removeClass("colorRed borderBlack");
```

```
$("#input").toggleClass("colorRed borderBlack");
```

	当	
	switch	
	是 true	
	时,添加	
toggleClass(classes, switch)	类,switch	<code>\$("#input").toggleClass("colorRed border Black", true);</code>
	ch 为	
	false	
	时,删除	
	类	

2.修改 CSS 样式

函数名	说明	例子
	访问	
	第一个	
css(name)	个	<code>\$("#input").css("color")</code>
	匹	
	配	
	元	
	素	

的
样
式
属
性
把
一
个
"
名
/
值
"
css(properties) 对
设
置
给
所
有
匹
配
元

```
$("#input").css({border:"solid 3px silver",color:"red"})
```

素的
的
样
式
属
性
为
匹
配
的
元
素
设
置
同
一
个
样
式
属
性
如

```
css(name,value  
)
```

```
$("#input").css("border-width","5");
```

果
是
数
字
,
将
自
动
转
换
为
像
素
值

最后附上一些获取常用的属性的方法:

1.宽、高相关

函数名	说明	例子
width()	获取第一个匹配元素的宽度,默认为 px	<code>\$("#txt1").width()</code>
width(val)	为匹配的元素设置宽度值,默认为 px	<code>\$("#txt1").width(200)</code>

height()	获取第一个匹配元素的高度,默认为 px	<code>\$("#txt1").height()</code>
height(val)	为匹配的元素设置宽度值,默认为 px	<code>\$("#txt1").height(20)</code>
innerWidth()	获取第一个匹配元素内部区域宽度(包括 padding,不包括 border)	<code>\$("#txt1").innerWidth()</code>
innerHeight()	获取第一个匹配元素内部区域高度(包括 padding,不包括 border)	<code>\$("#txt1").innerHeight()</code>
outerWidth([margin])	获取第一个匹配元素外部区域宽度(包括 padding,border margin 为 true 则包括 margin,否则不包括)	<code>\$("#txt1").outerWidth()</code>
outerHeight([margin])	获取第一个匹配元素外部区域高度(包括 padding,border margin 为 true 则包括 margin,否则不包括)	<code>\$("#txt1").outerHeight(true)</code>

括

padding,border)

margin 为 true 则包

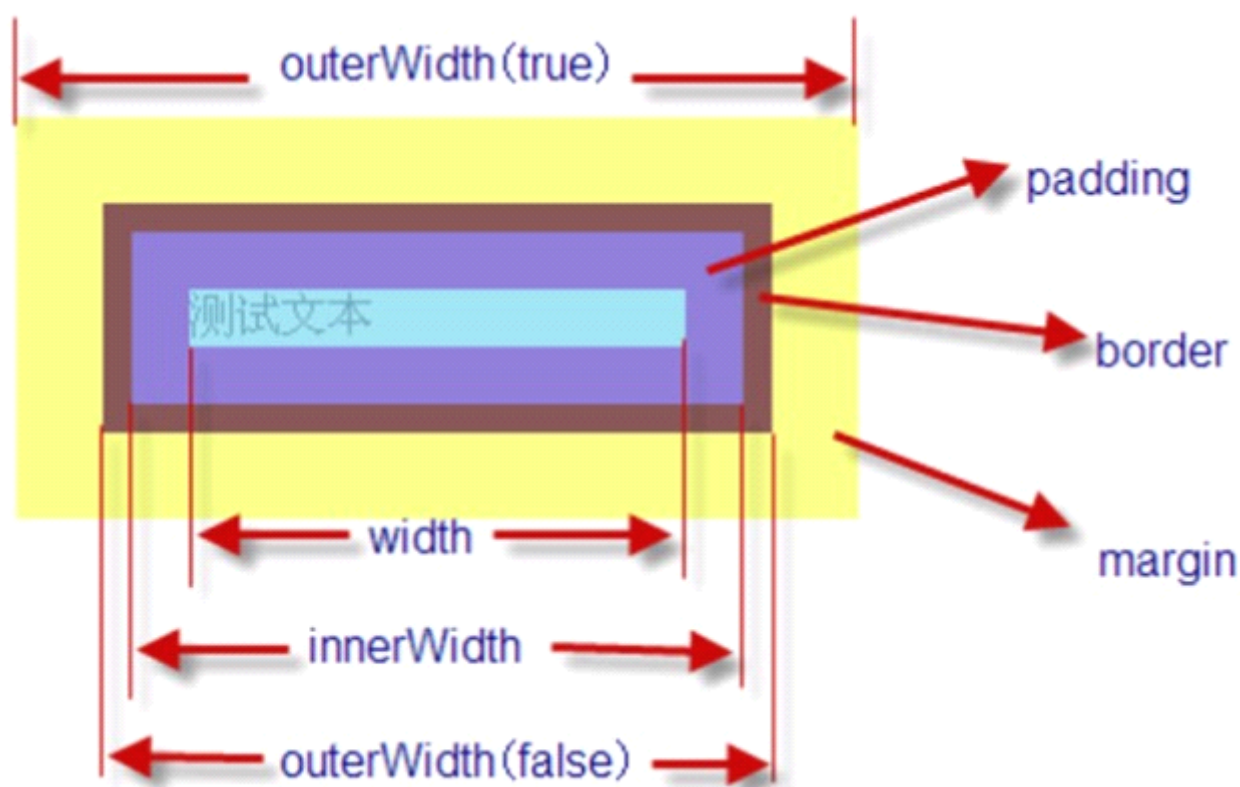
括 margin,否则不

包括

注意:

返回的高度、宽度均为数字,不带 px

参照一张图,会更容易理解些.



2.位置相关

在设计一些弹出对象的脚本中,经常需要动态获取弹出坐标并且设置元素的位置.jQuery 为我们提供了位置相关的各个函数.

函数名	说明	例子
	获取匹配元素在当前窗口的相对偏移	
<code>offset()</code>	偏移 只对可见元素有效	<code>\$("#btn").offset().top</code> <code>\$("#btn").offset().left</code>
	获取匹配元素相对父元素的偏移 只对可见元素有效	
<code>position()</code>	偏移	<code>\$("#btn").position().top</code> <code>\$("#btn").position().left</code>
	获取匹配元素相对滚动条顶部的偏移	
<code>scrollTop()</code>	偏移 对可见元素和隐藏元素均有效	<code>\$("#div").scrollTop()</code>
	设置垂直滚动条顶部偏移为该值 对可见元素和隐藏元素均有效	
<code>scrollTop(val)</code>	偏移	<code>\$("#div").scrollTop(200)</code>
	获取匹配元素相对滚动条左部的偏移	
<code>scrollLeft()</code>	偏移 对可见元素和隐藏元素均有效	<code>\$("#div").scrollLeft()</code>
	设置水平滚动条左侧的偏移 对可见元素和隐藏元素均有效	
<code>scrollLeft(val)</code>	偏移	<code>\$("#div").scrollLeft(200)</code>

注意:

`offset` 方法是相对于当前窗口的相对偏移,而 `position` 方法是相对于父元素的偏移

要特别注意区分 `attr()`和 `css()`, 比如说:

```
$comment_text.css("background-color","white");
$comment_text.attr("disabled",false);
```

（八十九）跟我学 jquery（五）jquery 中的 ajax 详解

Ajax 让用户页面丰富起来, 增强了用户体验. 使用 Ajax 是所有 Web 开发的必修课. 虽然 Ajax 技术并不复杂, 但是实现方式还是会因为每个开发人员的而有所差异. jQuery 提供了一系列 Ajax 函数来帮助我们统一这种差异, 并且让调用 Ajax 更加简单. Ajax 最常见的用法就是把一块 HTML 代码加载到页面的某个区域中去. 为此, 只需简单地选择所需的元素, 然后使用 `load()` 函数即可. 然后如果要用普通的 ajax 的话会使用大量的代码来实现. 下面我们就来看一下 jquery 中的 ajax。

一: jQuery Ajax 详解

jQuery 提供了几个用于发送 Ajax 请求的函数. 其中最核心也是最复杂的是 `jQuery.ajax(options)`, 所有的其他 Ajax 函数都是它的一个简化调用. 当我们想要完全控制 Ajax 时可以使用此结果, 否则还是使用简化方法如 `get`, `post`, `load` 等更加方便. 所以 `jQuery.ajax(options)` 方法放到最后一个介绍. 先来介绍最简单的 `load` 方法:

1. `load(url, [data], [callback])`: 载入远程 HTML 文件代码并插入至 DOM 中。

Returns: jQuery 包装集

`url (String)`: 请求的 HTML 页的 URL 地址。

`data (Map)`: (可选参数) 发送至服务器的 key/value 数据。

callback (Callback) : (可选参数) 请求完成时(不需要是 **success** 的)的回调函数。

说明:

load 方法能够载入远程 **HTML** 文件代码并插入至 **DOM** 中。

默认使用 **GET** 方式, 如果传递了 **data** 参数则使用 **Post** 方式.

- 传递附加参数时自动转换为 **POST** 方式。**jQuery 1.2** 中, 可以指定选择符, 来筛选载入的 **HTML** 文档, **DOM** 中将仅插入筛选出的 **HTML** 代码。语法形如 **"url #some > selector"**, 默认的选择器是 **"body>*"**。

讲解:

load 是最简单的 **Ajax** 函数, 但是使用具有局限性:

1 它主要用于直接返回 **HTML** 的 **Ajax** 接口

2 **load** 是一个 **jQuery** 包装集方法, 需要在 **jQuery** 包装集上调用, 并且会将返回的 **HTML** 加载到对象中, 即使设置了回调函数也还是会加载.

不过不可否认 **load** 接口设计巧妙并且使用简单. 下面通过示例来演示 **Load** 接口的使用:

[javascript] [view plaincopyprint?](#)

```
1. <script type="text/javascript" src="../../scripts/jquery-1.3.2-min.js"></script>
2. <script type="text/javascript">
3.     $(function()
4.     {
5.         $("#btnAjaxGet").click(function(event)
6.         {
7.             //发送 Get 请求
```

```

8.          $("#divResult").load("../data/AjaxGetMethod.aspx?param=btnAjaxGet_click" + "&tamp=" + (new Date()).getTime());
9.      });
10.     $("#btnAjaxPost").click(function(event)
11.     {
12.         //发送 Post 请求
13.         $("#divResult").load("../data/AjaxGetMethod.aspx", { "param": "btnAjaxPost_click" });
14.     });
15.     $("#btnAjaxCallBack").click(function(event)
16.     {
17.         //发送 Post 请求, 返回后执行回调函数.
18.         $("#divResult").load("../data/AjaxGetMethod.aspx", { "param": "btnAjaxCallBack_click" }, function(responseText, textStatus, XMLHttpRequest)
19.         {
20.             responseText = " Add in the CallBack Function! <br/>" + responseText
21.             $("#divResult").html(responseText); //或者: $(this).html(responseText);
22.         });
23.     });
24.     $("#btnAjaxFiltHtml").click(function(event)
25.     {
26.         //发送 Get 请求, 从结果中过滤掉 "鞍山" 这一项
27.         $("#divResult").load("../data/AjaxGetCityInfo.aspx?resultType=html" + "&tamp=" + (new Date()).getTime() + " ul>li:not(:contains('鞍山'))");
28.     });
29. })
30. </script>
31.</head>
32.<body>
33. <button id="btnAjaxGet">使用 Load 执行 Get 请求</button><br />
34. <button id="btnAjaxPost">使用 Load 执行 Post 请求</button><br />
35. <button id="btnAjaxCallBack">使用带有回调函数的 Load 方法</button><br />
36. <button id="btnAjaxFiltHtml">使用 selector 过滤返回的 HTML 内容</button>

```

```
37. <br />
38. <div id="divResult"></div>
39.</body>
40.</html>
```

上面的示例演示了如何使用 **Load** 方法.

提示:我们要时刻注意浏览器缓存, 当使用 **GET** 方式时要添加时间戳参数 `(new Date()).getTime()` 来保证每次发送的 URL 不同, 可以避免浏览器缓存.

提示: 当在 `url` 参数后面添加了一个空格, 比如 " " 的时候, 会出现 "无法识别符号" 的错误, 请求还是能正常发送. 但是无法加载 HTML 到 DOM. 删除后问题解决.

2. `jQuery.get(url, [data], [callback])`: 使用 **GET** 方式来进行异步请求

参数:

`url (String)`: 发送请求的 URL 地址.

`data (Map)`: (可选) 要发送给服务器的数据, 以 **Key/value** 的键值对形式表示, 会做为 **QueryString** 附加到请求 URL 中。

`callback (Function)`: (可选) 载入成功时回调函数(只有当 **Response** 的返回状态是 **success** 才是调用该方法)。

这是一个简单的 **GET** 请求功能以取代复杂 `$.ajax`。请求成功时可调用回调函数。如果需要在出错时执行函数, 请使用 `$.ajax`。示例代码:

[javascript] [view plaincopyprint?](#)

```

1. $.get("./Ajax.aspx", {Action:"get",Name:"caoshenghuan"}, function (data, textStatus)
    {
2.         //返回的 data 可以是 xmlDoc, jsonObj, html, text, 等等.
3.         this; // 在这里 this 指向的是 Ajax 请求的选项配置信息, 请参考下图
4.         alert(data);
5.         //alert(textStatus);//请求状态: success, error 等等。
6.         当然这里捕捉不到 error, 因为 error 的时候根本不会运行该回调函数
7.         //alert(this);
8.     });

```

点击发送请求:

jQuery.get()回调函数里面的 this , 指向的是 Ajax 请求的选项配置信息:

3. jQuery.post(url, [data], [callback], [type])

Returns: XMLHttpRequest

说明:

通过远程 HTTP POST 请求载入信息。

这是一个简单的 POST 请求功能以取代复杂 \$.ajax 。请求成功时可调用回调函数。如果需要在出错时执行函数, 请使用 \$.ajax。

讲解:

具体用法和 get 相同, 只是提交方式由"GET"改为"POST".

4. jQuery.getScript(url, [callback]) : 通过 GET 方式请求载入并执行一个 JavaScript 文件。

参数

url (String) : 待载入 JS 文件地址。

callback (Function) : (可选) 成功载入后回调函数。

jQuery 1.2 版本之前，**getScript** 只能调用同域 JS 文件。1.2 中，您可以跨域调用 JavaScript 文件。注意：Safari 2 或更早的版本不能在全局作用域中同步执行脚本。如果通过 **getScript** 加入脚本，请加入延时函数。这个方法可以用在例如当只有编辑器 **focus()** 的时候才去加载编辑器需要的 JS 文件。下面看一些示例代码：

加载并执行 **test.js**。jQuery 代码：

```
$.getScript("test.js");
```

加载并执行 **AjaxEvent.js**，成功后显示信息。

5. **jQuery.ajax(options)**

Returns: XMLHttpRequest

说明：

通过 HTTP 请求加载远程数据。

jQuery 底层 AJAX 实现。简单易用的高层实现见 **\$.get**, **\$.post** 等。

\$.ajax() 返回其创建的 XMLHttpRequest 对象。大多数情况下你无需直接操作该对象，但特殊情况下可用于手动终止请求。

\$.ajax() 只有一个参数：参数 **key/value** 对象，包含各配置及回调函数信息。详细参数选项见下。

注意：如果你指定了 **dataType** 选项，请确保服务器返回正确的 MIME 信息，(如 **xml** 返回 **"text/xml"**)。错误的 MIME 类型可能导致不可预知的错误。见 [Specifying the Data Type for AJAX Requests](#)。

注意：如果 `dataType` 设置为"`script`"，那么所有的远程(不在同一域名下)的 `POST` 请求都将转化为 `GET` 请求。(因为将使用 `DOM` 的 `script` 标签来加载)

jQuery 1.2 中，您可以跨域加载 `JSON` 数据，使用时需将数据类型设置为 `JSONP`。使用 `JSONP` 形式调用函数时，

如 "`myurl?callback=?`" jQuery 将自动替换 `?` 为正确的函数名，以执行回调函数。数据类型设置为 `"jsonp"` 时，jQuery 将自动调用回调函数。

讲解：

这是 jQuery 中 `Ajax` 的核心函数，上面所有的发送 `Ajax` 请求的函数内部最后都会调用此函数。`options` 参数支持很多参数，使用这些参数可以完全控制 `ajax` 请求。在 `Ajax` 回调函数中的 `this` 对象也是 `options` 对象。

因为平时使用最多的还是简化了的 `get` 和 `post` 函数，所以在此不对 `options` 参数做详细讲解了。`options` 参数文档请见：

<http://docs.jquery.com/Ajax/jQuery.ajax#options>

参数列表：

参数名	类型	描述
<code>url</code>	<code>String</code>	(默认：当前页地址) 发送请求的地址。
<code>type</code>	<code>String</code>	(默认：" <code>GET</code> ") 请求方式 (" <code>POST</code> " 或 " <code>GET</code> ")，默认为 " <code>GET</code> ". 注意：其它 <code>HTTP</code> 请求方法，如 <code>PUT</code> 和 <code>DELETE</code> 也可以使用，但仅部分浏

览器支持。

timeout	Number	设置请求超时时间（毫秒）。此设置将覆盖全局设置。
async	Boolean	(默认: true) 默认设置下，所有请求均为异步请求。如果需要发送同步请求，请将此选项设置为 false 。注意，同步请求将锁住浏览器，用户其它操作必须等待请求完成才可以执行。
beforeSend	Function	<p>发送请求前可修改 XMLHttpRequest 对象的函数，如添加自定义 HTTP 头。</p> <p>XMLHttpRequest 对象是唯一的参数。</p> <pre>function (XMLHttpRequest) { this; // the options for this ajax request }</pre>
cache	Boolean	(默认: true) jQuery 1.2 新功能，设置为 false 将不会从浏览器缓存中加载请求信息。
complete	Function	<p>请求完成后回调函数 (请求成功或失败时均调用)。参数: XMLHttpRequest 对象，成功信息字符串。</p> <pre>function (XMLHttpRequest, textStatus) { this; // the options for this ajax request }</pre>
contentType	String	(默认: "application/x-www-form-urlencoded") 发送信息至服务器时内容编码类型。默认值适合

大多数应用场合。

data

Object,
String

发送到服务器的数据。将自动转换为请求字符串格式。GET 请求中将附加在 URL 后。查看 processData 选项说明以禁止此自动转换。必须为 Key/Value 格式。如果为数组，jQuery 将自动为不同值对应同一个名称。如 {foo:["bar1", "bar2"]} 转换为 '&foo=bar1&foo=bar2'。

dataType

String

预期服务器返回的数据类型。如果不指定，jQuery 将自动根据 HTTP 包 MIME 信息返回 responseXML 或 responseText，并作为回调函数参数传递，可用值：

"xml": 返回 XML 文档，可用 jQuery 处理。

"html": 返回纯文本 HTML 信息；包含 script 元素。

"script": 返回纯文本 JavaScript 代码。不会自动缓存结果。

"json": 返回 JSON 数据。

"jsonp": JSONP 格式。使用 JSONP 形式调用函数时，如 "myurl?callback=?" jQuery 将自动替换 ? 为正确的函数名，以执行回调函数。

error	Function	<p>(默认: 自动判断 (xml 或 html)) 请求失败时将调用此方法。这个方法有三个参数:</p> <p>XMLHttpRequest 对象, 错误信息, (可能) 捕获的错误对象。</p> <pre>function (XMLHttpRequest, textStatus, errorThrown) { // 通常情况下 textStatus 和 errorThrown 只有 // 其中一个有值 this; // the options for this ajax request }</pre>
global	Boolean	<p>(默认: true) 是否触发全局 AJAX 事件。设置为 false 将不会触发全局 AJAX 事件, 如 ajaxStart 或 ajaxStop。可用于控制不同的 Ajax 事件</p>
ifModified	Boolean	<p>(默认: false) 仅在服务器数据改变时获取新数据。使用 HTTP 包 Last-Modified 头信息判断。</p>
processData	Boolean	<p>(默认: true) 默认情况下, 发送的数据将被转换为对象(技术上讲并非字符串) 以配合默认内容类型 "application/x-www-form-urlencoded"。如果要发送 DOM 树信息或其它不希望转换的信息, 请设置为 false。</p>
success	Function	<p>请求成功后回调函数。这个方法有两个参数: 服务</p>

器返回数据，返回状态

```
function (data, textStatus) {  
    // data could be xmlDoc, jsonObj, html, text, etc  
    ...  
    this; // the options for this ajax request  
}
```

这里有几个 Ajax 事件参数: beforeSend , success , complete , error 。

我们可以定义这些事件来很好的处理我们的每一次的 Ajax 请求。注意一下，

这些 Ajax 事件里面的 this 都是指向 Ajax 请求的选项信息的(请参考

说 get() 方法时的 this 的图片)。

请认真阅读上面的参数列表，如果你要用 jQuery 来进行 Ajax 开发，那么

这些参数你都必需熟知的。

示例代码，获取 CSDN 首页的文章题目：

[javascript] [view plain](#)[copy](#)[print?](#)

```
1. $.ajax({  
2.     type: "get",  
3.     url: "http://www.blog.csdn.net",  
4.     beforeSend: function(XMLHttpRequest){  
5.         //ShowLoading();  
6.     },  
7.     success: function(data, textStatus){  
8.         $(".ajax.ajaxResult").html("");  
9.         $(".item",data).each(function(i, domEle){  
10.            $(".ajax.ajaxResult").append("<li>"+$(domEle).children("title").text()+"</li>  
    >");  
11.        });  
12.    },  
13.    complete: function(XMLHttpRequest, textStatus){  
14.        //HideLoading();  
15.    },  
16.    error: function(){
```

```
17.         //请求出错处理
18.     }
19.});
```

二.Ajax 相关函数.

jQuery 提供了一些相关函数能够辅助 Ajax 函数.

1. jQuery.ajaxSetup(options)

无返回值

说明:

设置全局 AJAX 默认 options 选项。

讲解:

有时我们的希望设置页面上所有 Ajax 属性的默认行为.那么就可以使用此函数设置 options 选项, 此后所有的 Ajax 请求的默认 options 将被更改.

options 是一个对象, 可以设置的属性请此连接:

<http://docs.jquery.com/Ajax/jQuery.ajax#toptions>

比如在页面加载时, 我使用下面的代码设置 Ajax 的默认 option 选项:

[javascript] [view plain](#)[copy](#)[print?](#)

```
1. $.ajaxSetup({
2.     url: "../data/AjaxGetMethod.aspx",
3.     data: { "param": "ziqu.zhang" },
4.     global: false,
5.     type: "POST",
6.     success: function(data, textStatus) { $("#divResult").html(data); }
7. });
```

上面的代码设置了一个 Ajax 请求需要的基本数据：请求 url, 参数, 请求类型, 成功后的回调函数。

此后我们可以使用无参数的 `get()`, `post()` 或者 `ajax()` 方法发送 ajax 请求。完整的示例代码如下：

[javascript] [view plain](#)[copy](#)[print?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
   w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns="http://www.w3.org/1999/xhtml">
3. <head>
4.   <title>jQuery Ajax - Load</title>
5.   <script type="text/javascript" src="../scripts/jquery-1.3.2-vsdoc2.js"></script>
6.   <script type="text/javascript">
7.     $(document).ready(function()
8.     {
9.       $.ajaxSetup({
10.        url: "../data/AjaxGetMethod.aspx",
11.        data: { "param": "ziqu.zhang" },
12.        global: false,
13.        type: "POST",
14.        success: function(data, textStatus) { $("#divResult").html(data); }
15.      });
16.      $("#btnAjax").click(function(event) { $.ajax(); });
17.      $("#btnGet").click(function(event) { $.get(); });
18.      $("#btnPost").click(function(event) { $.post(); });
19.      $("#btnGet2").click(function(event) { $.get("../data/AjaxGetMethod.aspx",{ "
        param": "other" }); });
20.    });
21.  </script>
22.</head>
23.<body>
```



```

24. <button id="btnAjax">不传递参数调用 ajax()方法</button><br />
25. <button id="btnGet">不传递参数调用 get()方法</button><br />
26. <button id="btnPost">不传递参数调用 post()方法</button><br />
27. <button id="btnGet2">传递参数调用 get()方法, 使用全局的默认回调函数
    </button><br />
28. <br />
29. <div id="divResult"></div>
30.</body>
31.</html>

```

注意当使用 `get()` 或者 `post()` 方法时, 除了 `type` 参数将被重写为 "GET" 或者 "POST" 外, 其他参数只要不传递都是使用默认的全局 `option`. 如果传递了某一个选项, 比如最后一个按钮传递了 `url` 和参数, 则本次调用会以传递的选项为准. 没有传递的选项比如回调函数还是会使用全局 `option` 设置值.

2.serialize()

Returns: **String**

说明:

序列表表格内容为字符串, 用于 **Ajax** 请求。

序列化最常用在将表单数据发送到服务器端时. 被序列化后的数据是标准格式, 可以被几乎所有的而服务器端支持.

为了尽可能正常工作, 要求被序列化的表单字段都有 `name` 属性, 只有一个 `eid` 是无法工作的.

像这样写 `name` 属性:

```
<input id="email" name="email" type="text" />
```

讲解:

`serialize()`函数将要发送给服务器的 `form` 中的表单对象拼接成一个字符串, 便于我们使用 `Ajax` 发送时获取表单数据. 这和一个 `Form` 按照 `Get` 方式提交时, 自动将表单对象的名/值放到 `url` 上提交差不多.

比如这样一个表单:

生成的字符串

为: `single=Single¶m=Multiple¶m=Multiple3&check=check2&radio=radio1`

提示: 代码见 `chapter6\7-serialize.htm`

3.serializeArray()

Returns: `Array<Object>`

说明:

序列化表格元素 (类似 `'.serialize()'` 方法) 返回 `JSON` 数据结构数据。

注意, 此方法返回的是 `JSON` 对象而非 `JSON` 字符串。需要使用插件或者第三方库进行字符串化操作。

讲解:

看说明文档让我有所失望, 使用此函数获取到的是 `JSON` 对象, 但是 `jQuery` 中没有提供将 `JSON` 对象转化为 `JSON` 字符串的方法。

在 `JSON` 官网上没有找到合适的 `JSON` 编译器, 最后选用了 `jquery.json` 这个 `jQuery` 插件:

<http://code.google.com/p/jquery-json/>

使用起来异常简单:

[javascript] [view plain](#)[copy](#)[print?](#)

```
1. var thing = {plugin: 'jquery-json', version: 1.3};
2. var encoded = $.toJSON(thing);           /*{"plugin": "jquery-json", "version": 1.3}*/

3. var name = $.evalJSON(encoded).plugin;    /*"jquery-json"
4. var version = $.evalJSON(encoded).version; /* 1.3
```

使用 `serializeArray()` 再配合 `$.toJSON` 方法, 我们可以很方便的获取表单对象的 JSON, 并且转换为 JSON 字符串:

```
$("#results").html( $.toJSON( $("#form").serializeArray() ));
```

结果为:

```
[{"name": "single", "value": "Single"}, {"name": "param", "value": "Multiple"}, {"name": "param", "value": "Multiple3"}, {"name": "check", "value": "check2"}, {"name": "radio", "value": "radio1"}]
```

六.全局 Ajax 事件

在 `jQuery.ajaxSetup(options)` 中的 `options` 参数属性中, 有一个 `global` 属性:

global

类型:布尔值

默认值: true

说明:是否触发全局的 Ajax 事件.

这个属性用来设置是否触发全局的 **Ajax** 事件. 全局 **Ajax** 事件是一系列伴随 **Ajax** 请求发生的事件. 主要有如下事件:

名称	说明
<code>ajaxComplete(callback)</code>	AJAX 请求完成时执行函数
<code>ajaxError(callback)</code>	AJAX 请求发生错误时执行函数
<code>ajaxSend(callback)</code>	AJAX 请求发送前执行函数
<code>ajaxStart(callback)</code>	AJAX 请求开始时执行函数
<code>ajaxStop(callback)</code>	AJAX 请求结束时执行函数
<code>ajaxSuccess(callback)</code>	AJAX 请求成功时执行函数

用一个示例讲解各个事件的触发顺序:

[html] [view plaincopyprint?](#)

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
   w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2. <html xmlns=
   >
3. <head>
4.   <title>jQuery Ajax - AjaxEvent</title>
5.   <script type=
       src=
   ></script>
6.   <script type=
       >
7.     $(document).ready(function()
8.     {
9.       $("#btnAjax").bind("click", function(event)
10.      {
11.        $.get("../data/AjaxGetMethod.aspx");
12.      })
13.      $("#divResult").ajaxComplete(function(evt, request, settings) { $(this).appe
       nd('<div>ajaxComplete</div>'); })
```

```

14.      $("#divResult").ajaxError(function(evt, request, settings) { $(this).append('<div>ajaxError</div>'); })
15.      $("#divResult").ajaxSend(function(evt, request, settings) { $(this).append('<div>ajaxSend</div>'); })
16.      $("#divResult").ajaxStart(function() { $(this).append('<div>ajaxStart</div>');
    })
17.      $("#divResult").ajaxStop(function() { $(this).append('<div>ajaxStop</div>');
    })
18.      $("#divResult").ajaxSuccess(function(evt, request, settings) { $(this).append('<div>ajaxSuccess</div>'); })
19.      });
20.  </script>
21. </head>
22. <body>
23.  <br /><button id=          >发送 Ajax 请求 </button><br/>
24.  <div id=          ></div>
25. </body>
26. </html>

```

我们可以通过将默认 options 的 global 属性设置为 false 来取消全局 Ajax 事件的触发。

（九十）跟我学 jquery（六）jquery 中事件详解

由于 jquery 本身就是 web 客户端的有力帮手，所以事件对于它来说就显得尤为重要了，事件是脚本编程的灵魂. 所以此内容也是 jQuery 学习的重点。

在传统的 javascript 中，注册一个事件也是非常简单的事情，下面我们具体看一下一个简单的示例：

[html] [view plaincopyprint?](#)

```
1. document.getElementById("testDiv2").onclick =          ;
```

等效于:

[html] [view plaincopyprint?](#)

```
1. <div id=          onclick=          !!!");">单击事件 1</div>
```

注意两者的区别了吗? 我们常用的修改元素属性添加事件的方式, 实际上是建立了一个匿名函数:

[html] [view plaincopyprint?](#)

```
1. document.getElementById("testDiv1").onclick =          (event)
2. {
3.   alert("!!!");
4. };
```

这个匿名函数的签名和我们手写的 showMsg 签名相同, 所以可以把 showMsg 直接赋值给 onclick.

这种方式的弊端是:

1. 只能为一个事件绑定一个事件处理函数. 使用"="赋值会把前面为此时间绑定的所有事件处理函数冲掉.
2. 在事件函数(无论是匿名函数还是绑定的函数)中获取事件对象的方式在不同浏览器中要特殊处理:
3. 添加多播委托的函数在不同浏览器中是不一样的.

所以我们首先应该摒弃<div onclick="..."></div>这种通过修改元素属性添加事件的方式. 尽量使用添加多播事件委托的方式为一个事件绑定多个事件处理函数, 比如为 document 对象的单击事件添加一个关闭弹出层的方法, 使用多播就不会影响 document 对象原有的事件处理函数.

Jquery 事件:

从上面我们看到了 javascript 中注册事件的弊端了, 这些弊端真正避免起来也挺麻烦的, 所以 jquery 想到了这一点, 他几乎把 javascript 中的事件弊端解决到了极点, 我们可以很简单的实现我们在 javascript 中很麻烦才能实现的功能。正所谓有了 jQuery, 天天喝茶水. 下面是在 jQuery 中最常使用的 bind() 方法举例:

[html] [view plaincopyprint?](#)

```
1. $("#testDiv").bind("click", showMsg);
```

我们为 id 是 testDiv4 的元素, 添加列 click 事件的事件处理函数 showMsg.

使用 jQuery 事件处理函数的好处:

1. 添加的是多播事件委托. 也就是为 click 事件又添加了一个方法, 不会覆盖对象的 click 事件原有的事件处理函数.

[javascript] [view plaincopyprint?](#)

1. `$("#testDiv").bind("click", function(event) { alert("one"); });`
2. `$("#testDiv").bind("click", function(event) { alert("two"); });`

单击 testDiv 对象时, 依次提示"one"和"two".

2. 统一了事件名称.

添加多播事件委托时, ie 中是事件名称前面有"on". 但是使用 bind()函数我们不用区分 ie 和 dom, 因为内部 jQuery 已经帮我们统一了事件的名称.

3. 可以将对象行为全部用脚本控制.

让 HTML 代码部分只注意"显示"逻辑. 现在的趋势是将 HTML 的行为, 内容与样式切分干净. 其中用脚本控制元素行为, 用 HTML 标签控制元素内容, 用 CSS 控制元素样式. 使用 jQuery 事件处理函数可以避免在 HTML 标签上直接添加事件.

Jquery 常用事件函数:

虽然我们可以使用事件处理函数完成对象事件的几乎所有操作, 但是 jQuery 提供了对常用事件的封装. 比如单击事件对应的两个方法 click()和 click(fn)分别用来触发单击事件和设置单击事件.

设置单击事件:

[javascript] [view plaincopyprint?](#)

```
1. $("#testDiv").click(function(event) { alert("test div clicked ! "); });
```

等效于:

[javascript] [view plaincopyprint?](#)

```
1. $("#testDiv").bind("click", function(event) { alert("test div clicked ! "); });
```

触发单击事件:

[javascript] [view plaincopyprint?](#)

```
1. $("#testDiv").click();
```

等效于

[javascript] [view plaincopyprint?](#)

```
1. $("#testDiv").trigger("click");
```

注意这里等效的是 `trigger` 而不是 `triggerHandler`.

下面我们来看一下这些常用的事件函数

方法	描述
bind()	向匹配元素附加一个或更多事件处理器
blur()	触发、或将函数绑定到指定元素的 <code>blur</code> 事件
change()	触发、或将函数绑定到指定元素的 <code>change</code> 事件

click()	触发、或将函数绑定到指定元素的 click 事件
dblclick()	触发、或将函数绑定到指定元素的 double click 事件
delegate()	向匹配元素的当前或未来的子元素附加一个或多个事件处理器
die()	移除所有通过 live() 函数添加的事件处理程序。
error()	触发、或将函数绑定到指定元素的 error 事件
event.isDefaultPrevented()	返回 event 对象上是否调用了 event.preventDefault() 。
event.pageX	相对于文档左边缘的鼠标位置。
event.pageY	相对于文档上边缘的鼠标位置。
event.preventDefault()	阻止事件的默认动作。
event.result	包含由被指定事件触发的事件处理器返回的最后一个值。
event.target	触发事件的 DOM 元素。
event.timeStamp	该属性返回从 1970 年 1 月 1 日到事件发生时的毫秒数。
event.type	描述事件的类型。

event.which	指示按了哪个键或按钮。
focus()	触发、或将函数绑定到指定元素的 focus 事件
keydown()	触发、或将函数绑定到指定元素的 key down 事件
keypress()	触发、或将函数绑定到指定元素的 key press 事件
keyup()	触发、或将函数绑定到指定元素的 key up 事件
live()	触发、或将函数绑定到指定元素的 load 事件
load()	触发、或将函数绑定到指定元素的 load 事件
mousedown()	触发、或将函数绑定到指定元素的 mouse down 事件
mouseenter()	触发、或将函数绑定到指定元素的 mouse enter 事件
mouseleave()	触发、或将函数绑定到指定元素的 mouse leave 事件
mousemove()	触发、或将函数绑定到指定元素的 mouse move 事件

mouseout()	触发、或将函数绑定到指定元素的 mouse out 事件
mouseover()	触发、或将函数绑定到指定元素的 mouse over 事件
mouseup()	触发、或将函数绑定到指定元素的 mouse up 事件
one()	向匹配元素添加事件处理器。每个元素只能触发一次该处理器。
ready()	文档就绪事件（当 HTML 文档就绪可用时）
resize()	触发、或将函数绑定到指定元素的 resize 事件
scroll()	触发、或将函数绑定到指定元素的 scroll 事件
select()	触发、或将函数绑定到指定元素的 select 事件
submit()	触发、或将函数绑定到指定元素的 submit 事件
toggle()	绑定两个或多个事件处理器函数，当发生流的 click 事件时执行。
trigger()	所有匹配元素的指定事件
triggerHandler()	第一个被匹配元素的指定事件

unbind()	从匹配元素移除一个被添加的事件处理器
undelegate()	从匹配元素移除一个被添加的事件处理器, 现在或将来
unload()	触发、或将函数绑定到指定元素 的 unload 事件

交互帮助方法:

除了基本的实践, jQuery 提供了两个和事件相关的帮助方

法: **hover(over, out)** 和 **toggle(fn, fn2, fn3, fn4, ...)**

1. **hover(over, out)**

hover 函数主要解决在原始 javascript 中 **mouseover** 和 **mouseout** 函数存在的问题, 看下面这个示例:

有两个 **div**(红色区域), 里面分别嵌套了一个 **div**(黄色区域). HTML 代码如下:

[html] [view plaincopyprint?](#)

```

1. <div class=      id=      >
2.   Outer 1
3.   <div class=      id=      >Inner 1</div>
4. </div>
5. <div class=      id=      >
6.   Outer 2
7.   <div class=      id=      >Inner 2</div>
8. </div>
9. <div id=      ></div>

```

绑定如下事件:

[javascript] [view plaincopyprint?](#)

```
1. <script type="text/javascript">
2.     function report(event) {
3.         $('#console').append('<div>'+event.type+'</div>');
4.     }
5.     $(function(){
6.         $('#outer1')
7.             .bind('mouseover',report)
8.             .bind('mouseout',report);
9.         $('#outer2').hover(report,report);
10.    });
11. </script>
```

Outer1 我们使用了 `mouseover` 和 `mouseout` 事件, 当鼠标从 **Outer1** 的红色区域移动到黄色区域时, 会发现虽然都是在 `outer1` 的内部移动, 但是却触发了 `mouseout` 事件:

很多时候我们不希望出现上图的结果, 而是希望只有鼠标在 **Outer1** 内部移动时不触发事件, **Outer2** 使用 `Hover()` 函数实现了这个效果:

注意这里的事件名称进入叫做 `"mouseenter"`, 离开叫做 `"mouseleave"`, 而不再使用 `"mouseover"` 和 `"mouseleave"` 事件.

有经验的开发人员会立刻想到在制作弹出菜单时, 经常遇到这个问题: 为弹出菜单设置了 `mouseout` 事件自动关闭, 但是鼠标在弹出菜单内移动时常常莫名

其妙触发 mouseout 事件让菜单关闭. hover()函数帮助我们很好的解决了这个问题.

2. toggle(fn, fn2, fn3,fn4,...)

toggle 函数可以为对象添加 click 事件绑定函数, 但是设置每次点击后依次的调用函数。

如果点击了一个匹配的元素，则触发指定的第一个函数，当再次点击同一元素时，则触发指定的第二个函数，如果有更多函数，则再次触发，直到最后一个。随后的每次点击都重复对这几个函数的轮番调用。

可以使用 `unbind("click")`来删除。

下面的示例演示如何使用 toggle 函数:

[html] [view plaincopyprint?](#)

```
1. <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/T
   R/html4/strict.dtd">
2. <html>
3. <head>
4.   <title>toggle example</title>
5.   <link rel=           type=           href=           >
6.   <script type=           src=           ></script>
7.   <script type=           >
8.     $(function()
9.     {
10.      $("li").toggle(
11.        function()
12.        {
13.          $(this).css({ "list-style-type": "disc", "color": "blue" });
14.        },
15.        function()
16.        {
17.          $(this).css({ "list-style-type": "square", "color": "red" });
```

```

18.         },
19.         function()
20.         {
21.             $(this).css({ "list-style-type": "none", "color": "" });
22.         }
23.     );
24. })
25. </script>
26.</head>
27.<body>
28.    <ul>
29.        <li style=                >click me</li>
30.    </ul>
31.</body>
32.</html>

```

结果是每点击一次"click me"变换一次列表符号和文字颜色.

使用 jQuery 事件对象

使用事件自然少不了事件对象. 因为不同浏览器之间事件对象的获取, 以及事件对象的属性都有差异, 导致我们很难跨浏览器使用事件对象.

jQuery 中统一了事件对象, 当绑定事件处理函数时, 会将 jQuery 格式化后的事件对象作为唯一参数传入:

```
$("#testDiv").bind("click", function(event) { });
```

关于 event 对象的详细说明, 可以参考 jQuery 官方文

档: <http://docs.jquery.com/Events/jQuery.Event>

jQuery 事件对象将不同浏览器的差异进行了合并, 比如可以在所有浏览器中通过 `event.target` 属性来获取事件的触发者(在 IE 中使用原生的事件对象, 需要访问 `event.srcElement`).

下面是 jQuery 事件对象可以在浏览器支持的属性:

属性名称	描述	举例
<code>type</code>	事件类型.	<code>\$("#a").click(function(event) { alert(event.type); });</code>
	如果 使用 一个 事件 处理 函数	

来处理多个事件,可以使用此属性获得事件类型,比如

	click	
target	.	
	获取事件触发者DOM对象	<pre> \$("a[href=http://google.com]").click(function(event) { alert(event.target.href); }); </pre>
data	事件调用时传入额外参数	<pre> \$("a").each(function(i) { \$(this).bind('click', {index:i}, function(e){ alert('my index is ' + e.data.index); }); }); </pre>

```
relatedTarget 对 $("a").mouseout(function(event) {  
get           于 alert(event.relatedTarget);  
              });
```

鼠

标

事

件,

标

示

触

发

事

件

时

离

开

或

者

进

入

的

DOM

元

	素	
currentTa	冒	\$("#p").click(function(event) {
rget	泡	alert(event.currentTarget.nodeName);
		});
	前	结果:P
	的	
	当	
	前	
	触	
	发	
	事	
	件	
	的	
	DO	
	M	
	对	
	象,	
	等	
	同	
	于	
	this.	
pageX/Y	鼠	\$("#a").click(function(event) {
		alert("Current mouse position: " + event.pageX + ", "
	标	+ event.pageY);
		});
	事	

件
中,
事
件
相
对
于
页
面
原
点
的
水
平/
垂
直
坐
标.

result 上 一 个 事
\$("p").click(function(event) {
 return "hey"
});
\$("p").click(function(event) {
 alert(event.result);
});

件 结果:"hey"

处

理

函

数

返

回

的

值

```
timeStamp 事件 var last;  
p          件  $("p").click(function(event) {  
            if( last )  
                alert( "time since last event " + event.timeStamp -  
                last );  
            last = event.timeStamp;  
            });
```

时

的

时

间

戳.

上面是 jQuery 官方文档中提供的 event 对象的属性. 在"jQuery 实战"一书中还提供了下面的多浏览器支持的属性, 时间关系我没有尝试每一个属性, 大家可以帮忙验证是否在所有浏览器下可用:

属性名称 描述

举

altKey	Alt 键是否被按下. 按下返回 true
ctrlKey	ctrl 键是否被按下, 按下返回 true
metaKey	Meta 键是否被按下, 按下返回 true. meta 键就是 PC 机器的 Ctrl 键,或者 Mac 机器上面的 Command 键
shiftKey	Shift 键是否被按下, 按下返回 true
keyCode	对于 keyup 和 keydown 事件返回被按下的键. 不区分大小写, a 和 A 都返回 65. 对于 keypress 事件请使用 which 属性, 因为 which 属性跨浏览时依然可靠.
which	对于键盘事件, 返回触发事件的键的数字编码. 对于鼠标事件, 返回鼠标按键号(1 左,2 中,3 右).
screenX/Y	对于鼠标事件, 获取事件相对于屏幕原点的水平/垂直坐标

事件对象除了拥有属性, 还拥有事件. 有一些是一定会用到的事件比如取消冒泡 `stopPropagation()` 等. 下面是 jQuery 事件对象的函数列表:

名称	说明	举例
preventDefault()	取消可能引起任何语意操作的事	<pre>\$("#a").click(function(event){ event.preventDefault(); // do something });</pre>

件. 比如<a>元素
 的 href 链接加
 载, 表单提交以及
 click 引起复选框
 的状态切换.

isDefaultPrevented()	是否调用过 preventDefault() 方法	<pre> \$("a").click(function(event){ alert(event.isDefaultPrevent ed()); event.preventDefault(); alert(event.isDefaultPrevent ed()); }); </pre>
stopPropagation()	取消事件冒泡	<pre> \$("p").click(function(event){ event.stopPropagation(); // do something }); </pre>
isPropagationStopp ed()	是否调用过 stopPropagation() 方法	<pre> \$("p").click(function(event){ alert(event.isPropagationSt opped()); event.stopPropagation(); alert(event.isPropagationSt opped()); }); </pre>
stopImmediatePropag ation()	取消执行其他的 事件处理函数并 取消事件冒泡. 如果同一个事件 绑定了多个事件 处理函数, 在其中 一个事件处理函	<pre> \$("p").click(function(event){ event.stopImmediatePropag ation(); }); \$("p").click(function(event){ // This function won't be exe cuted }); </pre>

数中调用此方法

后将不会继续调

用其他的事件处

理函数.

isImmediatePropaga tionStopped()	是否调用过	\$("#p").click(function(event){
		alert(event.isImmediateProp
stopImmediatePr	agationStopped());	
opagation()	event.stopImmediatePropag	
方法	ation();	
	alert(event.isImmediateProp	
	agationStopped());	
	});	

这些函数中 stopPropagation() 是我们最长用的也是一定会用到的函数. 相当于操作原始 event 对象的 event.cancelBubble=true 来取消冒泡.

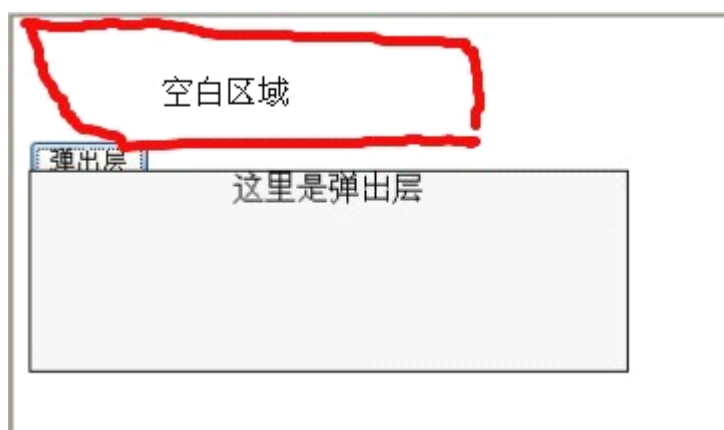
（九十一）跟我学 jquery（七）jquery 动画大体验

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

最近一直感觉自己好忙，每天都浑浑噩噩的过着，转眼间，好像有好长时间没有更新笨鸟到菜鸟了。现在想想，实在罪过了。自从从北京回来就一直投入不了状态，所以也想利用一下这个写博客的机会来促进自己早日投入状态吧。今天我们要讲的是 jquery 动画的东西，其实一说到动画，我相信很多人想到的应该是 flash 吧。其实不然，现在我们将要用 jquery 来改变 flash 的一统天下，当然，我们这里所说的动画和 flash 所做的动画还是有很大的区别的。所以也谈不上他两个产品有什么竞争关系，下面我们就先通过一个简单的实例来看一下 jquery 的强大吧。

在很多 web 开发过程中，我们都会用到弹出层的情况，我们再做弹出层的时候大多数都应该用的 js 实现把。假设有如下需求：



- 单击图中的"显示提示文字"按钮,在按钮的下方显示一个弹出
- 单击任何空白区域或者弹出层,弹出层消失

用原始 javascript 我们也完全可以完成这个工作. 有以下几点注意事项:

- 1.弹出层的位置需要动态计算.因为触发弹出事件的对象可能出现在页面的任何位置,比如截图中的位置.
- 2.为 document 绑定单击是关闭弹出层的函数,要使用多播委托,否则可能冲掉其他人在 document 绑定的函数.
- 3.为 document 绑定了关闭函数后,需要在显示函数中取消事件冒泡,否则弹出层将显示后立刻关闭

下面我们来看一下用 jquery 来实现这里的弹出层是怎么实现的:

[html] [view plain](#)[copy](#)[print?](#)

```
1. <html xmlns="http://www.w3.org/1999/xhtml" >
2.
3. <head>
4.
5. <title>jQuery – 弹出层动画</title>
6.
7. <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js" ></script>
8.
9. <script type="text/javascript">
10.
11.     $(document).ready(function()
12.
13.     {
14.
15.         //动画速度
16.
17.         var speed = 500 ;
18.
```

```

19.      //绑定事件处理
20.
21.      $("#btnShow").click(function(event)
22.
23.      {
24.
25.          //取消事件冒泡,所谓事件冒泡就是在子控件触发事件时父控件也触发相应
            的事件
26.
27.          event.stopPropagation();
28.
29.          //获得触发事件控件的位置
30.
31.          var offset = $(event.target).offset();
32.
33.          //设置弹出层的位置
34.
35.          $("#divPop").css({ top: offset.top + $(event.target).height() + "px", left: of
            fset.left });
36.
37.          //动画显示弹出层
38.
39.          $("#divPop").show(speed);
40.
41.      });
42.
43.      //单击空白区域隐藏弹出层
44.
45.      $(document).click(function(event) { $("#divPop").hide(speed) });
46.
47.      //单击弹出层则自身隐藏
48.
49.      $("#divPop").click(function(event) { $("#divPop").hide(speed) });
50.
51.      });

```

```

52.
53. </script>
54.
55. </head>
56.
57. <body>
58.
59. <div>
60.
61. <br /><br /><br />
62.
63. <button id=          >弹出层</button>
64.
65. </div>
66.
67. <!-- 弹出层 -->
68.
69. <div id=          style="background-color: #f0f0f0; border: solid 1px #000000; p
    osition: absolute; display:none;
70.
71.     width: 300px; height: 100px;">
72.
73. <div style=          >这里是弹出层</div>
74.
75. </div>
76.
77. </body>
78.
79. </html>

```

从上边代码的效果我们可以看到，利用 jquery 除了实现了自动隐藏和弹出层，还实现了动画弹出的效果。这样的效果让我这个初学 jquery 的小菜确

实欢喜。所以说感觉很有必要专门抽出一篇来讲解一下这个 jquery 动画。Ok, 下面我们就一起来正式看一下 jquery 动画的东西

jQuery 的动画函数主要分为三类:

基本动画函数:既有透明度渐变,又有滑动效果。是最常用的动画效果函数。

滑动动画函数:仅使用滑动渐变效果。

淡入淡出动画函数:仅使用透明度渐变效果。

这三类动画函数效果各不相同,用法基本一致.大家可以自己尝试.另外也许上面的三类函数效果都不是我们想要的,那么 jQuery 也提供了 **自定义动画函数**, 将控制权放在我们手里让我们自己定义动画效果。下面我们就来一一看一下这三种动画函数。

一.基本动画函数:

1、show()

显示隐藏匹配元素。这个就是 'show(speed, [callback])'无动画的版本。如果选择的元素是可见的,这个方法将不会改变任何东西。无论这个元素是通过hide()方法隐藏的还是在CSS里设置了display:none;,这个方法都将有效。

例如: 显示所有段落, \$("p").show()

2、show(speed,[callback])

以优雅的动画显示匹配的元素,并且在显示完成后可选择返回一个回调函。可根据指定的速度动态改变每个匹配元素高度、宽度和不透明度。

例如: 用缓慢的动画将隐藏的段落显示出来, 历时 600 毫秒,

```
$("#p").show(600)
```

3、hide()

隐藏显示元素。这个就是 'hide(speed, [callback])'的无动画版。如果选择的元素是隐藏的，这个方法将不会改变任何东西。

例如：隐藏所有段落，`$("#p").hide()`

4、hide(speed,[callback])

以优雅的动画隐藏所有匹配的元素，并在显示完成后可选的触发一个回调函数。可以根据指定的速度动态地改变每个匹配元素的高度、宽度和不透明度。在 jQuery 1.3 中，padding 和 margin 也会有动画，效果更流畅。

例如：用 600ms 的时间将段落缓慢的隐藏，`$("#p").hide("slow");`

5、toggle

切换元素的可见状态。如果元素时可见的，切换为隐藏的；如果元素是隐藏的，切换为可见的。

例如：切换所有段落的可见状态，`$("#p").toggle()`

6、toggle(switch)

根据 switch 参数切换元素的可见状态(true 为可见,false 为隐藏)。如果 switch 设为 true,则调用 show()方法来显示匹配的元素，如果 switch 设为 false 则调用 hide()来隐藏元素。

例如：切换所有段落的可见状态，var
`flip=0;$("#button").click(function(){$("#p").toggle(flip++%2==0);});`

7、toggle(speed,[callback])

以优雅的动画切换所有匹配的元素，并在显示完成后可选的触发一个回调函数。可根据指定的速度动态的改变每个匹配元素的高度、宽度和不透明

度。jquery 1.3,padding 和 margin 也会有动画,效果更流畅。

例如：用 200ms 将段落迅速切换显示状态，之后弹出一个对话框，
`$("#p").toggle("fast",function(){alert("hello !");});`

说明： speed 参数可以使用三种预定速度之一的字符串("slow", "normal", or "fast")或表示动画时长的毫秒数值(如： 1000).

二.滑动动画函数

基本动画函数的效果是一个综合了滑动和透明度渐变的函数, jQuery 还单独提供了只有滑动效果的相关函数.

名称	说明	举例
slideDown(speed, [callback])	通过高度变化（向下增大）来动态地显示所有匹配的元素，在显示完成后可选地触发一个回调函数。 这个动画效果只调整元	用 600 毫秒缓慢的将段落滑下： <code>\$("#p").slideDown("slow");</code>

	<p>素的高度，可以使匹配的</p> <p>元素以“滑动”的方式显示</p> <p>出来。在</p> <p>jQuery 1.3</p> <p>中，上下的</p> <p>padding 和</p> <p>margin 也会有动画，效果更流畅。</p>	
<p>slideUp(speed, [callback])</p>	<p>通过高度变化（向上减小）来动态地</p> <p>隐藏所有匹配的元素，在</p> <p>隐藏完成后</p> <p>可选地触发一个回调函数。</p>	<p>用 600 毫秒缓慢的将段落滑上：</p> <pre>\$("p").slideUp("slow");</pre>
<p>slideToggle(speed, [callback])</p>	<p>通过高度变化来切换所</p>	<p>用 600 毫秒缓慢的将段落滑上或滑下：</p> <pre>\$("p").slideToggle("slow");</pre>

	有匹配元素的可见性，并在切换完成后可选地触发一个回调函数。	
--	-------------------------------	--

说明：

slideDown 就是 show 的滑动效果版本，slideUp 就是 hide 的滑动效果版本，slideToggle 就是 toggle 的滑动效果版本。

参数完全相同：

```
$("#divPop").slideDown(200);
$("#divPop").slideUp("fast");
$("#divPop").slideToggle("slow");
```

三.淡入淡出动画函数

淡入淡出函数只提供透明度渐变的效果。

名称	说明	举例
fadeIn(speed, [callback])	通过不透明度的变化来实现所有匹配元素的淡入效果，并在动画完成后可选地触发一个回调函数。	用 600 毫秒缓慢的将段落淡入： <pre>\$("#p").fadeIn("slow");</pre>

	<p>这个动画只调整元素的不透明度，也就是说所有匹配的元素的高度和宽度不会发生变化。</p>	
fadeOut(speed, [callback])	<p>通过不透明度的变化来实现所有匹配元素的淡出效果，并在动画完成后可选地触发一个回调函数。</p>	<p>用 600 毫秒缓慢的将段落淡出：</p> <pre>\$("#p").fadeOut("slow");</pre>
fadeTo(speed, opacity, [callback])	<p>把所有匹配元素的不透明度以渐进方式调整到指定的不透明度，并在动画完成后可选地触发一个回调函数。</p>	<p>用 600 毫秒缓慢的将段落的透明度调整到 0.66，大约 2/3 的可见度：</p> <pre>\$("#p").fadeTo("slow", 0.66);\$("#p").fadeTo("slow",</pre>

四、自定义动画函数 **Custom**

1、animate(params,[duration],[easing],[callback])用于创建自定义动画的函

数。这个函数的关键在于制定动画形式及结果样式属性对象。这个对象中每个属性都表示一个可以变化的样式属性（如 height、top 或 opacity）。注意：所有指定的属性必须用骆驼形式，比如用 marginLeft 代替 margin-left。而每个属性的值表示这个样式属性到多少是动画结束。如果是一个数值，样式属性就会从当前的值渐变到指定的值。如果使用的是 hide、show、toggle 这样的字符串值，则会就该属性调用默认的动画形式。

例如：点击按钮后 div 元素的几个不同属性一同变化，

[html] [view plaincopyprint?](#)

```
1. $("#go").click(function(){
2.     $("#block").animate({
3.         width:"90%",height:"100%",fontSize:"10em",borderWidth:10
4.     },1000);
5. });
```

2、stop([clearQueue],[gotoEnd])

停止所有在指定元素上正在运行的动画。如果队列中有等待执行的动画（并且 clearQueue 没有设为 true），他们将被马上执行 clearQueue(Boolean):如果设置成 true，则清空队列。可以立即结束动画。gotoEnd (Boolean):让当前正在执行的动画立即完成，并且重设 show 和 hide 的原始样式，调用回调函数等。

例如：点击 Go 之后开始动画,点 Stop 之后会在当前位置停下来:

[html] [view plaincopyprint?](#)

```
1. //开始动画
```

```
2.    $("#go").click(function(){
3.    $(".block").animate({left: '+200px'}, 5000);
4.    });
5.    //当点击按钮后停止动画
6.    $("#stop").click(function(){
7.    $(".block").stop();
8.    });
```

（九十二）深入 java 虚拟机（一）——java 虚拟机底层结构详解

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

在以前的博客里面，我们介绍了在 java 领域中大部分的知识点，从最基础的 java 最基本语法到 SSH 框架。这里面应该包含了在 java 领域里面的大部分内容了吧。但是，那些知识点是让我们从一个应用的层面上了解了 java，java 程序真正底层的运行机制和一些底层虚拟机的工作我们还不了解，虽然这些内容在我们真正的开发中几乎用不到这些底层的東西，但对于我们对 java 的理解会有比较大的帮助。尤其也对以后 java 开发中的性能优化有很大帮助，可以使我們减少一些没必要的内存浪费等好处。所以，从今天开始，我将和大家一起来学习一下 java 虚拟机的内容。从底层开一下 java 的运行机制。

Java 虚拟机

Java 虚拟机 (Java Virtual Machine) 简称 JVM Java 虚拟机是一个想象中的机器，在实际的计算机上通过软件模拟来实现。Java 虚拟机有自己想象中的硬件，如处理器、堆栈、寄存器等，还具有相应的指令系统。下面我们来看一下这几部分比较重要的 java 虚拟机的结构

JVM 寄存器

所有的 CPU 均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义较多的寄存器，便可以从中得到更多的信息而不必对栈或内存进行访问，这有利于提高运行速度。然而，如果虚拟机中的寄存器比实际 CPU 的寄存器多，在实现虚拟机时就会占用处理器大量的时间来用常规存储器模

拟寄存器，这反而会降低虚拟机的效率。针对这种情况，JVM 只设置了 4 个最为常用的寄存器。它们是：**pc** 程序计数器，**optop** 操作数栈顶指针，**frame** 当前执行环境指针，**vars** 指向当前执行环境中第一个局部变量的指针，所有寄存器均为 32 位。**pc** 用于记录程序的执行。**optop,frame** 和 **vars** 用于记录指向 Java 栈区的指针。

JVM 栈结构

作为基于栈结构的计算机，Java 栈是 JVM 存储信息的主要方法。当 JVM 得到一个 java 字节码应用程序后，便为该代码中一个类的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架包括以下三类信息：局部变量执行环境操作数栈 局部变量用于存储一个类的方法中所用到的局部变量。**vars** 寄存器指向该变量表中的第一个局部变量。执行环境用于保存解释器对 Java 字节码进行解释过程中所需的信息。它们是：上次调用的方法、局部变量指针和操作数栈的栈顶和栈底指针。执行环境是一个执行一个方法的控制中心。例如：如果解释器要执行 **iadd**（整数加法），首先要从 **frame** 寄存器中找到当前执行环境，而后便从执行环境中找到操作数栈，从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。操作数栈用于存储运算所需操作数及运算的结果。

JVM 碎片回收堆

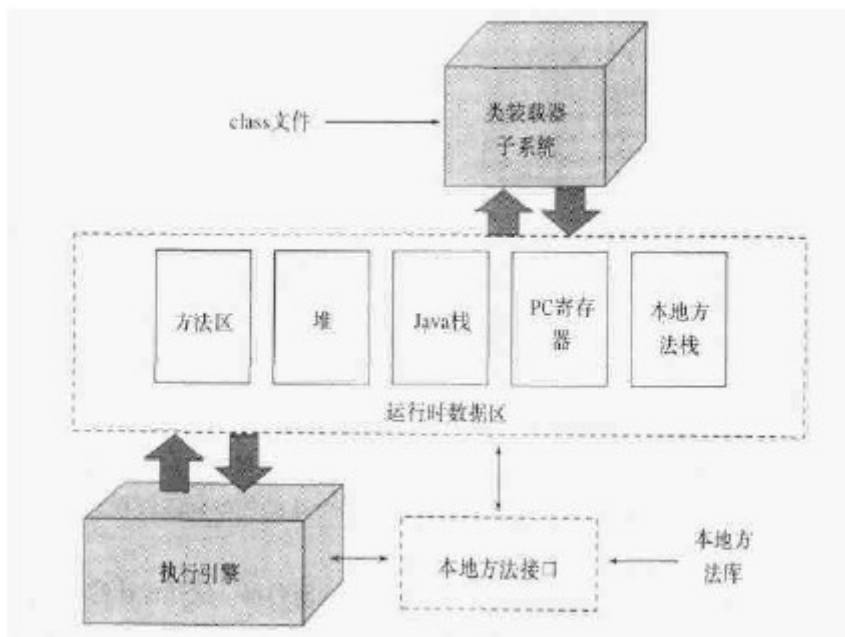
Java 类的实例所需的存储空间是在堆上分配的。解释器具体承担为类实例分配空间的工作。解释器在为一个实例分配完存储空间后，便开始记录对该实例所占用的内存区域的使用。一旦对象使用完毕，便将其回收到堆中。在 Java 语言中，除了 **new** 语句外没有其他方法为一对象申请和释放内存。对内存进

行释放和回收的工作是由 **Java** 运行系统承担的。这允许 **Java** 运行系统的设计者自己决定碎片回收的方法。在 **SUN** 公司开发的 **Java** 解释器和 **Hot Java** 环境中，碎片回收用后台线程的方式来执行。这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。

JVM 存储区

JVM 有两类存储区：常量缓冲池和方法区。常量缓冲池用于存储类名称、方法和字段名称以及串常量。方法区则用于存储 **Java** 方法的字节码。对于这两种存储区域具体实现方式在 **JVM** 规格中没有明确规定。这使得 **Java** 应用程序的存储布局必须在运行过程中确定，依赖于具体平台的实现方式。**JVM** 是为 **Java** 字节码定义的一种独立于具体平台的规格描述，是 **Java** 平台独立性的基础。目前的 **JVM** 还存在一些限制和不足，有待于进一步的完善，但无论如何，**JVM** 的思想是成功的。对比分析：如果把 **Java** 原程序想象成我们的 **C++** 原程序，**Java** 原程序编译后生成的字节码就相当于 **C++** 原程序编译后的 **80x86** 的机器码（二进制程序文件），**JVM** 虚拟机相当于 **80x86** 计算机系统，**Java** 解释器相当于 **80x86CPU**。在 **80x86CPU** 上运行的是机器码，在 **Java** 解释器上运行的是 **Java** 字节码。Java 解释器相当于运行 **Java** 字节码的“**CPU**”，但该“**CPU**”不是通过硬件实现的，而是用软件实现的。**Java** 解释器实际上就是特定的平台下的一个应用程序。只要实现了特定平台下的解释器程序，**Java** 字节码就能通过解释器程序在该平台下运行，这是 **Java** 跨平台的根本。当前，并不是在所有的平台下都有相应 **Java** 解释器程序，这也是 **Java** 并不能在所有的平台下都能运行的原因，它只能在已实现了 **Java** 解释器程序的平台下运行。

Java 虚拟机的体系结构图



Java 虚拟机从启动到结束的生命周期，当 java 虚拟机启动后，在如下几种情况下，Java 虚拟机将结束生命周期：

1. 执行了 `System.exit()` 方法
2. 程序正常执行结束
3. 程序在执行过程中遇到了异常或错误而异常终止
4. 由于操作系统出现错误而导致 Java 虚拟机进程终止

Java 虚拟机的栈有三个区域:局部变量区、运行环境区、操作数区。

局部变量区

每个 Java 方法使用一个固定大小的局部变量集。它们按照与 vars 寄存器的字偏移量来寻址。局部变量都是 32 位的。长整数和双精度浮点数占据了两个局部变量的空间,却按照第一个局部变量的索引来寻址。(例如,一个具有索引 n 的局部变量,如果是一个双精度浮点数,那么它实际占据了索引 n 和 n+1

所代表的存储空间)虚拟机规范并不要求在局部变量中的 **64** 位的值是 **64** 位对齐的。虚拟机提供了把局部变量中的值装载到操作数栈的指令,也提供了把操作数栈中的值写入局部变量的指令。

运行环境区

在运行环境中包含的信息用于动态链接,正常的方法返回以及异常捕捉。

操作数栈区

机器指令只从操作数栈中取操作数,对它们进行操作,并把结果返回到栈中。选择栈结构的原因是:在只有少量寄存器或非通用寄存器的机器(如 **Intel486**)上,也能够高效地模拟虚拟机的行为。操作数栈是 **32** 位的。它用于给方法传递参数,并从方法接收结果,也用于支持操作的参数,并保存操作的结果。例如,**iadd** 指令将两个整数相加。相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进堆栈的。这两个整数将从堆栈弹出、相加,并把结果压回到操作数栈中。

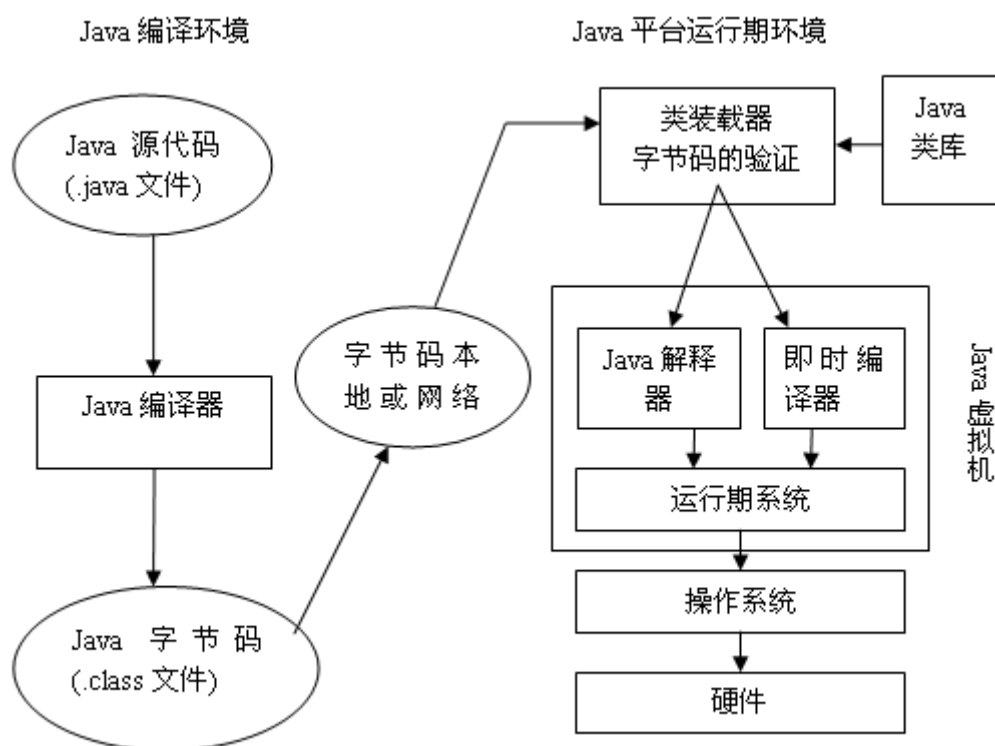
每个原始数据类型都有专门的指令对它们进行必须的操作。每个操作数在栈中需要一个存储位置,除了 **long** 和 **double** 型,它们需要两个位置。操作数只能被适用于其类型的操作符所操作。例如,压入两个 **int** 类型的数,如果把它们当作是一个 **long** 类型的数则是非法的。在 **Sun** 的虚拟机实现中,这个限制由字节码验证器强制实行。但是,有少数操作(操作符 **dupe** 和 **swap**),用于对运行时数据区进行操作时是不考虑类型的。

本地方法栈, 当一个线程调用本地方法时, 它就不再受到虚拟机关于结构和安全限制方面的约束, 它既可以访问虚拟机的运行期数据区, 也可以使用本地处理器以及任何类型的栈。例如, 本地栈是一个 **C** 语言的栈, 那么当 **C** 程

序调用 **C** 函数时，函数的参数以某种顺序被压入栈，结果则返回给调用函数。

在实现 **Java** 虚拟机时，本地方法接口使用的是 **C** 语言的模型栈，那么它的本地方法栈的调度与使用则完全与 **C** 语言的栈相同。

下图可以表示出来 **java** 程序运行的一个全过程



3 Java 虚拟机的运行过程

上面对虚拟机的各个部分进行了比较详细的说明，下面通过一个具体的例子来分析它的运行过程。

虚拟机通过调用某个指定类的方法 **main** 启动，传递给 **main** 一个字符串数组参数，使指定的类被装载，同时链接该类所使用的其它的类型，并且初始化它们。例如对于程序：

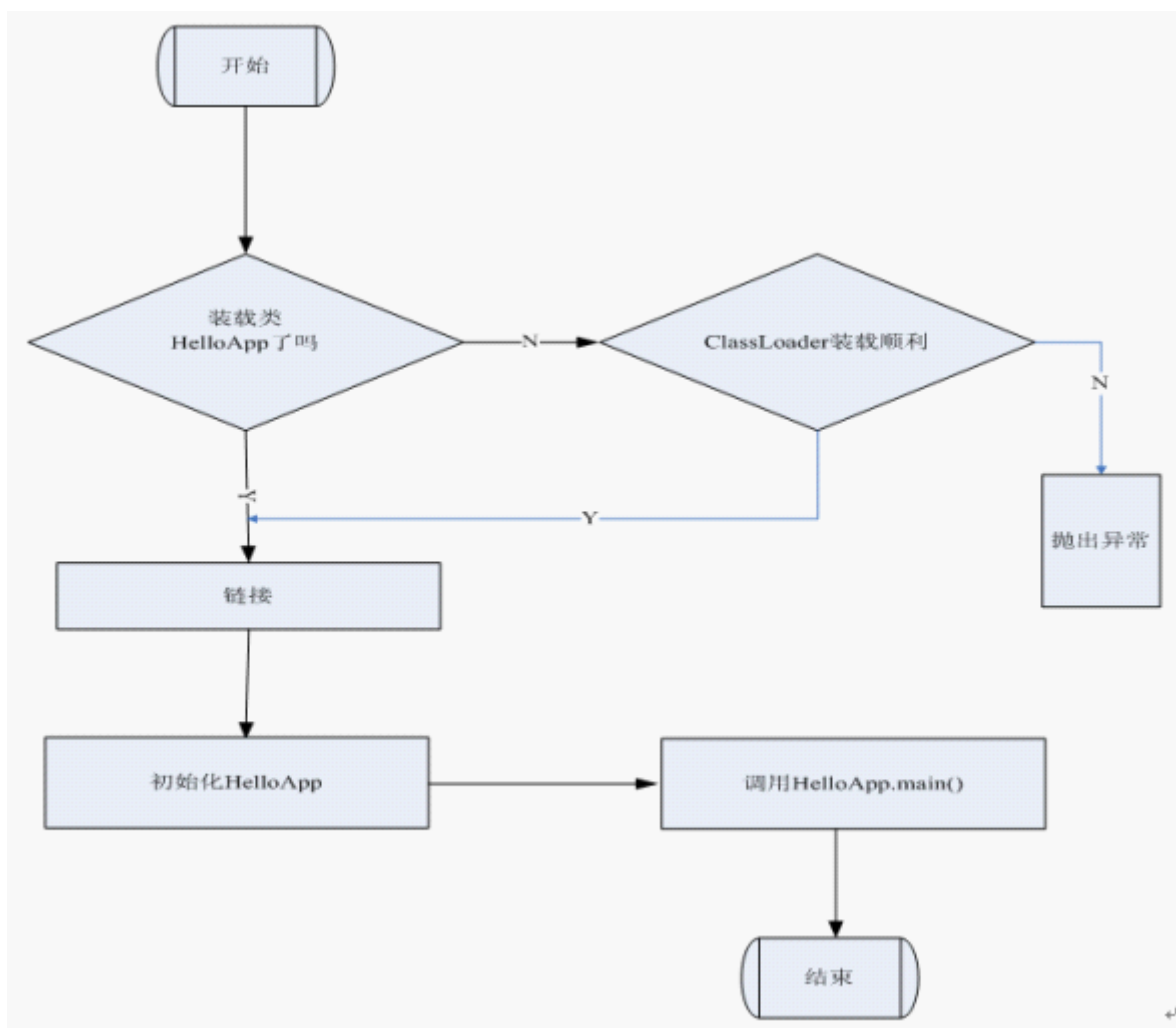
```
class HelloApp
```

```
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World!");  
        for (int i = 0; i < args.length; i++ )  
        {  
            System.out.println(args[i]);  
        }  
    }  
}
```

编译后在命令行模式下键入：`java HelloApp run virtual machine`

将通过调用 **HelloApp** 的方法 **main** 来启动 **java** 虚拟机，传递给 **main** 一个包含三个字符串"run"、"virtual"、"machine"的数组。现在我们略述虚拟机在执行 **HelloApp** 时可能采取的步骤。

开始试图执行类 **HelloApp** 的 **main** 方法，发现该类并没有被装载，也就是说虚拟机当前不包含该类的二进制代表，于是虚拟机使用 **ClassLoader** 试图寻找这样的二进制代表。如果这个进程失败，则抛出一个异常。类被装载后同时在 **main** 方法被调用之前，必须对类 **HelloApp** 与其它类型进行链接然后初始化。链接包含三个阶段：检验，准备和解析。检验检查被装载的主类的符号和语义，准备则创建类或接口的静态域以及把这些域初始化为标准的默认值，解析负责检查主类对其它类或接口的符号引用，在这一步它是可选的。类的初始化是对类中声明的静态初始化函数和静态域的初始化构造方法的执行。一个类在初始化之前它的父类必须被初始化。整个过程如下：



推荐阅读（内含 jvm 内存区域说明）：

Java 程序员从笨鸟到菜鸟之（九十三）深入 **java** 虚拟机（二）——
类加载器详解（上）

参考资料：<http://www.kuqin.com/java/20080525/8907.html>

（九十三）深入 java 虚拟机（二）——类加载器详解（上）

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

类加载器，顾名思义，类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：
Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 `java.lang.Class` 类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 `newInstance()` 方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。基本上所有的类加载器都是 `java.lang.ClassLoader` 类的一个实例。其实我们研究类加载器主要研究的就是类的生命周期

首先来了解一下 jvm（java 虚拟机）中的几个比较重要的内存区域，这几个区域在 java 类的生命周期中扮演着比较重要的角色：

方法区：在 java 的虚拟机中有一块专门用来存放已经加载的类信息、常量、静态变量以及方法代码的内存区域，叫做方法区。

常量池：常量池是方法区的一部分，主要用来存放常量和类中的符号引用等信息。

堆区：用于存放类的对象实例。

栈区：也叫 java 虚拟机栈，是由一个一个的栈帧组成的后进先出的栈式结构，栈帧中存放方法运行时产生的局部变量、方法出口等信息。当调用一个方法时，虚拟机栈中就会创建一个栈帧存放这些数据，当方法调用完成时，栈帧消失，如果方法中调用了其他方法，则继续在栈顶创建新的栈帧。

类的生命周期

当我们编写一个 java 的源文件后，经过编译会生成一个后缀名为 class 的文件，这种文件叫做字节码文件，只有这种字节码文件才能够在 java 虚拟机中运行，java 类的生命周期就是指一个 class 文件从加载到卸载的全过程。一个 java 类的完整的生命周期会经历加载、连接、初始化、使用、和卸载五个阶段，当然也有在加载或者连接之后没有被初始化就直接被使用的情况，这里我们主要来研究类加载器所执行的部分，也就是加载，链接和初始化。如图所示：



下面我先简单看一下类加载器所执行的三部分的简单介绍

1、加载：查找并加载类的二进制数据

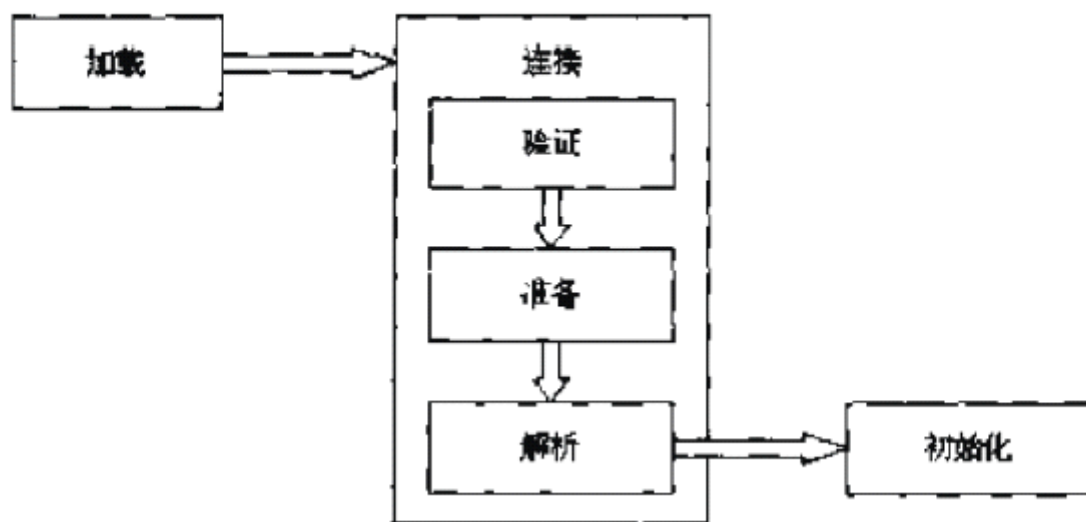
2、连接

- 验证：确保被加载的类的正确性
- 准备：为类的静态变量分配内存，并将其初始化为默认值
- 解析：把类中的符号引用转换为直接引用

3、初始化：为类的静态变量赋予正确的初始值

从上边我们可以看出类的静态变量赋了两回值。这是为什么呢？原因是，在连接过程中时为静态变量赋值为默认值，也就是说，只要是你定义了静态变量，不管你开始给没给它设置，我系统都为他初始化一个默认值。到了初始化过程，系统就检查是否用户定义静态变量时有没有给设置初始化值，如果有就把静态变量设置为用户自己设置的初始化值，如果没有还是让静态变量为初始化值

类的加载、连接和初始化



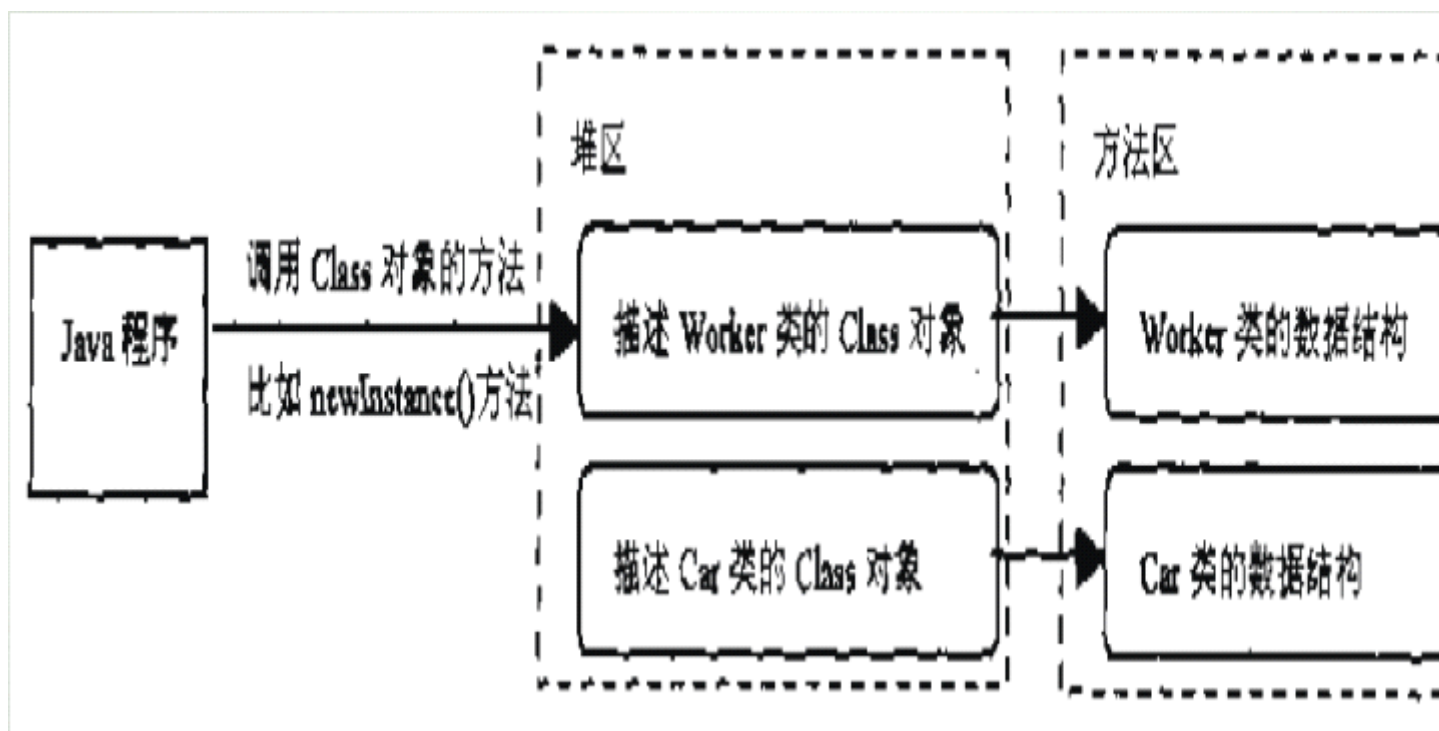
类的加载

类的加载指的是将类的.class 文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 `java.lang.Class` 对象，用来封装类在方法区内的数据结构。这里的 `class` 对象其实就像一面镜子一样，外面是类的源程序，里面是 `class` 对象，它实时的反应了类的数据结构和信息。

加载.class 文件的方式

- 1、从本地系统中直接加载
- 2、通过网络下载.class 文件
- 3、从 zip, jar 等归档文件中加载.class 文件
- 4、从专有数据库中提取.class 文件
- 5、将 Java 源文件动态编译为.class 文件

类的加载过程



结论：

- 1、类的加载的最终产品是位于堆区中的 **Class** 对象
- 2、**Class** 对象封装了类在方法区内的数据结构，并且向 **Java** 程序员提供了访问方法区内的数据结构的接口

Java 虚拟机给我们提供了两种类加载器：

1、**Java** 虚拟机自带的加载器

- 1) 根类加载器（使用 **C++** 编写，程序员无法在 **Java** 代码中获得该类）
- 2) 扩展加载器，使用 **Java** 代码实现
- 3) 系统加载器（应用加载器），使用 **Java** 代码实现

2、用户自定义的类加载器

`java.lang.ClassLoader` 的子类

用户可以定制类的加载方式

我们看一下 API 对 **ClassLoader** 的介绍：

类加载器是负责加载类的对象。**ClassLoader** 类是一个抽象类。如果给定类的二进制名称，那么类加载器会试图查找或生成构成类定义的数据。一般策略是将名称转换为某个文件名，然后从文件系统读取该名称的“类文件”。每个 **class** 对象都包含一个对定义它的 **ClassLoader** 的引用。

我们再来看一下 **Class** 类的一个方法 `getClassLoader`

```
public ClassLoader getClassLoader()
```

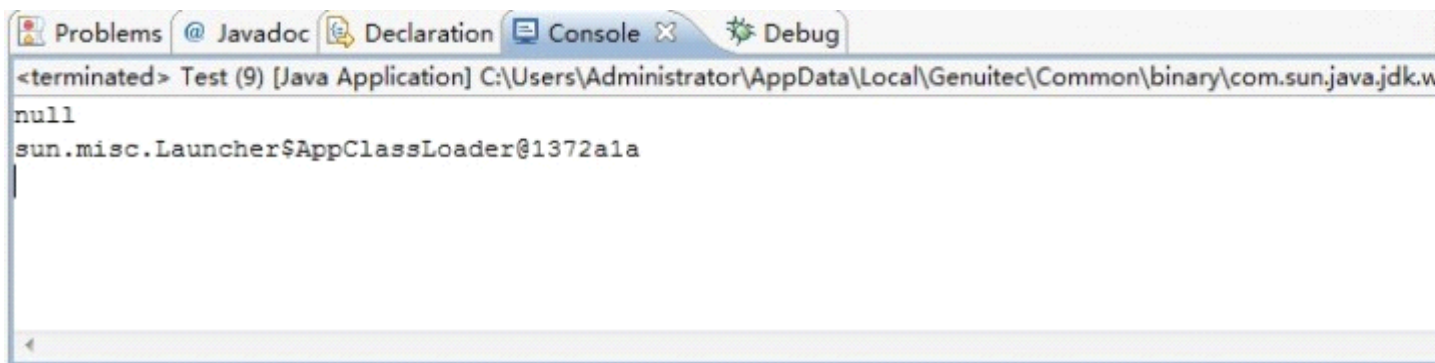
返回该类的类加载器。有些实现可能使用 `null` 来表示根类加载器。如果该类由根类加载器加载，则此方法在这类实现中将返回 `null`。

下面我们来看一个小例子来验证一下：

[java] view plaincopyprint?

```
1. package com.bzu.csh;
2. public class Test
3. {
4.     public static void main(String[] args) throws Exception
5.     {
6.         Class clazz = Class.forName("java.lang.String");
7.         System.out.println(clazz.getClassLoader());
8.         Class clazz2 = Class.forName("com.bzu.csh.ABC");
9.         System.out.println(clazz2.getClassLoader());
10.    }
11. }
12. class ABC
13. {
14. }
```

看一下打印结果，一目了然：



从上面打印结果可以看出，第一个为 `null`，也就是它用根类加载器加载的，第二个是我们自己写的类，也就是说，我们自己写的那个类用 `sun.misc.Launcher$AppClassLoader@1372a1a` 加载器加载的，我们可以看得到 **APP**，也就是应用类加载器，也就是系统加载器

还记得我们以前用过的动态代理吧，`InvocationHandler`，当我们利用 `proxy` 对象调用 `newProxyInstance` 建立一个代理类时，我们要给他传一个 **ClassLoader**，也就是类加载器，如下：

[java] [view plaincopyprint?](#)

```
1. public static Object newProxyInstance(ClassLoader loader,
2.                                     Class<?>[] interfaces,
3.                                     InvocationHandler h)
4.                                     throws IllegalArgumentException
```

当时我们学习的时候，只知道这里的 **loader** 随便给他设置一个类的类加载器就可以。现在我们来想想为什么这里需要一个类加载器呢？我们知道这个 `newProxyInstance` 是动态的给我们生成一个代理类，然后根据这个代理类生成一个代理对象。动态生成这个代理类之后我们不得把他加载到内存里吗，加载到内存里我们才可以用他。用什么加载到内存里，只有类加载器，所以我们要给他指定一个类加载器。

类加载器并不需要等到某个类被“首次主动使用”时再加载它。JVM 规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了 `.class` 文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（`LinkageError` 错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。大家在做 web 开发的时候可能会出现这种问题，比如我们在做测试的时候是用的 `jdk1.6`，而我们在部署的时候我们用的是 `jdk1.5`。这时候就很可能汇报 `LinkageError` 错误，版本不兼容。

类的连接

类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

验证： 当一个类被加载之后，必须要验证一下这个类是否合法，比如这个类是不是符合字节码的格式、变量与方法是不是有重复、数据类型是不是有效、继承与实现是否合乎标准等等。总之，这个阶段的目的就是保证加载的类是能够被 jvm 所运行。很多人都感觉，既然这个类都通过编译加载到内存里了，那肯定就是合法的了，为什么还要验证呢，这是因为这里的验证时为了避免有人恶意编写 class 文件，也就是说并不是通过编译得到的 class 文件。所以这里验证其实是检查的 class 文件的内部结构是否符合字节码的要求

准备： 准备阶段的工作就是为类的静态变量分配内存并设为 jvm 默认的初值，对于非静态的变量，则不会为它们分配内存。有一点需要注意，这时候，静态变量的初值为 jvm 默认的初值，而不是我们在程序中设定的初值。jvm 默认的初值是这样的：

基本类型（int、long、short、char、byte、boolean、float、double）的默认值为 0。

引用类型的默认值为 null。

常量的默认值为我们程序中设定的值，比如我们在程序中定义

`final static int a = 100`，则准备阶段中 a 的初值就是 100。

解析： 这一阶段的任务就是把常量池中的符号引用转换为直接引用。那么什么是符号引用，什么又是直接引用呢？我们来举个例子：我们要找一个人，

我们现有的信息是这个人的身份证号是 1234567890。只有这个信息我们显然找不到这个人，但是通过公安局的身份系统，我们输入 1234567890 这个号之后，就会得到它的全部信息：比如山东省滨州市滨城区 18 号张三，通过这个信息我们就能找到这个人了。这里，123456790 就好比是一个符号引用，而山东省滨州市滨城区 18 号张三就是直接引用。在内存中也是一样，比如我们要在内存中找一个类里面的一个叫做 show 的方法，显然是找不到。但是在解析阶段，jvm 就会把 show 这个名字转换为指向方法区的的一块内存地址，比如 c17164，通过 c17164 就可以找到 show 这个方法具体分配在内存的哪一个区域了。这里 show 就是符号引用，而 c17164 就是直接引用。在解析阶段，jvm 会将所有的类或接口名、字段名、方法名转换为具体的内存地址。

下一篇文章继续讲解类加载器内容。。。。

参考文章：<http://www.2cto.com/kf/201204/129386.html>

（九十四）深入 java 虚拟机（三）——类加载器（中）类的初始化

上接深入 java 虚拟机——[深入 java 虚拟机（二）——类加载器详解（上）](#)，在上一篇文章中，我们讲解了类的生命周期的加载和连接，这一篇我们接着上面往下看。

类的初始化:在类的生命周期执行完加载和连接之后就开始了类的初始化。在类的初始化阶段，java 虚拟机执行类的初始化语句，为类的静态变量赋值，在程序中，类的初始化有两种途径：（1）在变量的声明处赋值。（2）在静态代码块处赋值，比如下面的代码，a 就是第一种初始化，b 就是第二种初始化

[\[html\] view plaincopyprint?](#)

```
1. public class Test
2. {
3.     public static int a = ;
4.     public static int b ;
5.     static{
6.         b= ;
7.     }
8. }
```

静态变量的声明和静态代码块的初始化都可以看做静态变量的初始化，类的静态变量的初始化是有顺序的。顺序为类文件从上到下进行初始化，想到这，想起来[一个很无耻的面试题](#)，分享给大家看一下：

[\[java\] view plaincopyprint?](#)


```
1. package com.bzu.csh;
2. class Singleton
3. {
4.     private static Singleton singleton = new Singleton();
5.     public static int counter1;
6.     public static int counter2 = 0;
7.     private Singleton()
8.     {
9.         counter1++;
10.        counter2++;
11.    }
12.    public static Singleton getInstance()
13.    {
14.        return singleton;
15.    }
16. }
17. public class Test
18. {
19.     public static void main(String[] args)
20.     {
21.         Singleton singleton = Singleton.getInstance();
22.         System.out.println("counter1 = " + singleton.counter1);
23.         System.out.println("counter2 = " + singleton.counter2);
24.     }
25. }
```

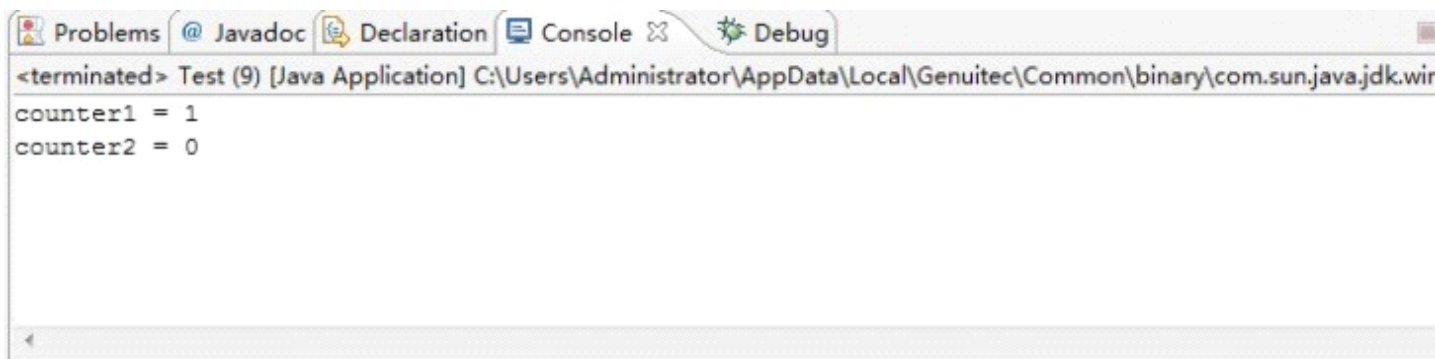
大家先看看这里的程序会输出什么？

不知道大家的答案是什么，如果不介意的话可以把你的答案写到评论上，看看有多少人的答案和你一样的。我先说说我刚开始的答案吧。我认为会输出：

counter1 = 1

Counter2 = 1

不知道大家的答案是不是这个，反正我的是。下面我们来看一下正确答案：



```
<terminated> Test (9) [Java Application] C:\Users\Administrator\AppData\Local\Genuitec\Common\binary\com.sun.java.jdk.win
counter1 = 1
counter2 = 0
```

不知道你做对没有，反正我刚开始做错了。好，现在我来解释一下为什么会是这个答案。在给出解释之前，我们先来看一个概念：

Java 程序对类的使用方式可分为两种

主动使用

被动使用

•所有的 Java 虚拟机实现必须在每个类或接口被 Java 程序“首次主动使用”时才初始化他们

主动使用（六种）

–创建类的实例

–访问某个类或接口的静态变量，或者对该静态变量赋值

–调用类的静态方法

–反射（如 `Class.forName(“com.bzu.csh.Test”)`）

–初始化一个类的子类

–Java 虚拟机启动时被标明为启动类的类（Java Test）

OK，我们开始解释一下上面的答案，程序开始运行，首先执行 main 方法，执行 main 方法第一条语句，调用 Singleton 类的静态方法，这里调用 Singleton 类的静态方法就是主动使用 Singleton 类。所以开始加载 Singleton 类。在加载 Singleton 类的过程中，首先对静态变量赋值为默认值，

```
Singleton=null
```

```
counter1 = 0
```

```
Counter2 = 0
```

给他们赋值完默认值之后，要进行的就是对静态变量初始化，对声明时已经赋值的变量进行初始化。我们上面提到过，初始化是从类文件从上到下赋值的。所以首先给 Singleton 赋值，给它赋值，就要执行它的构造方法，然后执行 counter1++;counter2++;所以这里的 counter1 = 1;counter2 = 1;执行完这个初始化之后，然后执行 counter2 的初始化，我们声明的时候给他初始化为 0 了，所以 counter2 的值又变为了 0.初始化完之后执行输出。所以这是的

```
counter1 = 1
```

```
counter2 = 0
```

类初始化步骤

- (1) 假如一个类还没有被加载或者连接，那就先加载和连接这个类
- (2) 假如类存在直接的父类，并且这个父类还没有被初始化，那就先初始化直接的父类
- (3) 假如类中存在初始化语句，那就直接按顺序执行这些初始化语句

在上边我们说了 java 虚拟机实现必须在每个类或接口被 **Java** 程序“首次主动使用”时才初始化他们，上面也举出了六种主动使用的说明。除了上述六种情形，其他使用 **Java** 类的方式都被看作是被动使用，不会导致类的初始化。程序中对子类的“主动使用”会导致父类被初始化；但对父类的“主动”使用并不会导致子类初始化（不可能说生成一个 **Object** 类的对象就导致系统中所有的子类都会被初始化）

当 **java** 虚拟机初始化一个类时，要求它的所有的父类都已经被初始化，但这条规则并不适用于接口。

在初始化一个类时，并不会先初始化它所实现的接口

在初始化一个接口时，并不会先初始化它的父接口

因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化。只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用。如果是调用的子类的父类属性，那么子类不会被初始化。

附录

一：基于 SSH 实现的简单学生选课系统（附源码）

首先声明：本小项目是因老师而做，主要为了完成老师项目要求，当然也添加了许多老师没有要求的功能，本项目代码非常简单，分享出来时为了和我一样的小菜鸟共同学习一下，只适合初学者拿来学习，大牛可直接绕过。如有不妥之处，欢迎大家提出意见

本项目为学生选课系统，下面附一下老师的主要要求：

1. *用户登录
2. *查看个人信息
3. *修改个人信息
4. *查看个人选课情况

5. *查看所有课程信息，能够选定课程
6. *退选课程
7. DIV+CSS 美化页面
8. 添加查询功能：如根据课程名进行模糊查询，课程开课学期进行查询
9. 国际化处理
10. 添加分页功能

当然，我在这基础之上添加了一些额外功能，本系统一共历时四天的时间完成，希望分享出来可以帮助大家学习，下面附一下 **DAO** 层的主要代码功能：

3.1通用数据库访问类 **HibernateUtil**

方法名	功能描述
add	添加对象
delete	删除对象
Update	更新对象
Select	查询对象
<u>Check</u>	验证登陆信息
selectPage	分页查询

3.2 **StudentDAO** 接口

方法名	功能描述
checkStuExists	判断所要添加的学号是否存在
exists	获得对应的页码的数据集合

pageList	获得对应 id 的学生对象
getStudent	修改学生信息
update	删除学生
<u>delete</u>	添加学生

3.3 StudentDAOImpl 实现类:

方法名	功能描述
checkStudent	判断所要添加的学号是否存在
exists	获得对应的页码的数据集合
pageList	获得对应 id 的学生对象
getStudent	修改学生信息
update	删除学生
<u>delete</u>	添加学生

3.4 CourseDAO 接口

方法名	功能描述
pageList	获得分页查询的当前页的结果
getCourse	获得对应 id 的课程对象
getCourses	获得学生的选课的集合
delete	删除课程
updateCourse	修改课程信息
<u>addCourse</u>	添加课程

3.5CourseDAOImpl 实现类:

方法名	功能描述
pageList	获得分页查询的当前页的结果
getCour	获得对应 id 的课程对象
getCourses	获得学生的选课集合
delete	删除课程
updateCourse	修改课程信息
addCourse	添加课程

3.6AdminerDAO 接口

方法名	功能描述
checkAdminer	验证登陆的管理员信息是否正确
exists	判断添加的管理员的用户名是否存在
addAdmin	添加管理员
getStu	获得对应 id 的学生对象
update	更新管理员信息

3.7AdminerDAOImpl 实现类:

方法名	功能描述
checkAdminer	验证登陆的管理员信息是否正确
exists	判断添加的管理员的用户名是否存在
addAdmin	添加管理员
getStu	获得对应 id 的学生对象

update

更新管理员信息

四、项目模块介绍

1.前台模块

主要功能：

1.1学生登陆

1.2个人信息管理

1.2.1:个人信息显示

1.2.2: 个人信息修改

1.3选课信息管理

1.3.1已选课程列表

1.4课表信息管理

1.4.1课表显示

下面看一下几张前台模块运行效果：

登陆界面：



修改个人信息

修改个人信息

学号	<input type="text" value="123"/>
密码	<input type="password" value="..."/>
姓名	<input type="text" value="曹胜欢"/>
性别	<input type="text" value="1"/>
生日	<input type="text" value="1990-8-22"/>
专业	<input type="text" value="软件技术"/>
上传头像	<input type="button" value="选择文件"/> 未选择文件
<div><input type="button" value="提交"/> <input type="button" value="重置"/></div>	

已选课程列表

操作	课程号	课程名	开课学期	课
	22	高数	22	2
	4	java啊	1	5

课程列表

查询条件

课程名: 课程号: 开课学期: 课时: 从

课程学分:

操作	课程号	课程名	开课学期
<input checked="" type="checkbox"/>	1	java	4
<input checked="" type="checkbox"/>	2	c#	1
<input checked="" type="checkbox"/>	4	java啊	1
<input checked="" type="checkbox"/>	19	计算机	1111
<input checked="" type="checkbox"/>	20	Flash	2
<input checked="" type="checkbox"/>	21	photoshop	2
<input checked="" type="checkbox"/>	22	高数	22
<input checked="" type="checkbox"/>	23	英语	2
<input checked="" type="checkbox"/>	24	.net高级	2
<input checked="" type="checkbox"/>	25	语文	2

整体页面显示:



学生选课系统



用户: 曹胜欢

注销

- 个人信息管理
- 选课信息管理
- ▼ 选课信息
 - 课程列表
 - 待续
 - 待续
 - 待续
 - 待续
 - 待续

查询条件

课程名: 课程号: 开课学期: 课时: 从 到

课程学分:

操作	课程号	课程名	开课学期
<input checked="" type="checkbox"/>	1	java	4
<input checked="" type="checkbox"/>	2	c#	1
<input checked="" type="checkbox"/>	4	java啊	1
<input checked="" type="checkbox"/>	19	计算机	1111
<input checked="" type="checkbox"/>	20	Flash	2
<input checked="" type="checkbox"/>	21	photoshop	2
<input checked="" type="checkbox"/>	22	高数	22
<input checked="" type="checkbox"/>	23	英语	2
<input checked="" type="checkbox"/>	24	.net高级	2
<input checked="" type="checkbox"/>	25	语文	2

第1/2页 首页 1 2 尾页 第0页

2后台管理模块

2.1管理员登录功能

2.2管理员信息显示

2.3添加管理员：这里用到了 **ajax** 异步验证技术来验证用户名是否存在

2.4更新管理员信息

2.5学生列表

2.6学生添加

2.7课程列表

2.8课程添加

添加管理员

添加管理员

用户名	<input type="text" value="123"/>	
密码	<input type="password"/>	
姓名	<input type="text"/>	
<input type="button" value="提交"/>		<input type="button" value="重置"/>

学生列表




















查询条件

姓名:

学号:

专业:

出生日期: 从

操作		学号	姓名	性别	生日
		1023110910	曹胜欢	1	1990-08-25
		1023110902	曹胜欢	1	1990-08-25
		1023110901	曹胜欢	0	1990-08-25
		1023110903	曹胜欢	1	1990-08-25
		1023110904	曹胜欢	0	1990-08-25
		1023110905	小金	0	1990-08-25
		1023110906	曹胜欢	1	1990-08-25
		1023110907	曹胜欢	1	1990-08-25
		1023110910	曹胜欢	1	1990-08-25
		1023110902	曹胜欢	1	1990-08-25

第1/4页

首页 1 2 3 4 尾页

第0 页

Go

利用模式窗口形式修改学生信息

操作		学号	姓名	性别	生日
		1023110910	曹胜欢	1	1990-08-25
		1023110902	曹胜欢	1	1990-08-25

localhost:8080/StudentSelectCourse/jsp/behind/StudentActionUp...

localhost:8080/StudentSelectCourse/jsp/behind/StudentAction!updateStuDialog?st

学生管理

学号	1023110902
密码	*****
姓名	曹胜欢
性别	1
生日	1990-08-25
专业	软件技术
上传头像	<div>选择文件 未选择文件</div>
<div>提交 重置</div>	

总体后台页面显示



最后一个功能就是在用户首页，用户可以选择是按管理员登陆还是学生登陆：

如下图



用力点击：[下载源码](#)

二：基于 SSH 的商场管理系统(附源码)

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

首先声明：本小项目也是因老师而做，但这个没有按老师说的做，我按我自己的想法来做的，相对于前段时间，这个小项目的业务逻辑相对于那个要复杂一些。本项目代码非常简单，分享出来时为了和我一样的小菜鸟共同学习一下，只适合初学者拿来学习，大牛可直接绕过。如有不妥之处，欢迎大家提出意见。这个程序是半个多月前做的，由于最近比较忙，所以现在才拿出来给大家分享

随着超市里货物种类和数量的大量增加，超市工作人员的工作量也随之增多，然而，日益繁重的工作使同志们日益疲惫，每位同志都在超负荷的运转，为出现工作失误制造了一定的有利条件，对于此，超市的管理层看在眼里，急在心理。怎样既可加快办事效率，又能减少工作失误，更好服务于社会主义四个现代化建设的问题，逐渐的进入到了领导的视线里，经过同志们以三个代表为指导思想，同心同德，集思广益，最终，在超市领导深思熟虑后果断决定近期上马一套为本超市量身定做的管理软件，它的上马将大大的提升本超市的工作管理水平，使员工们能更好的投入到工作中去。

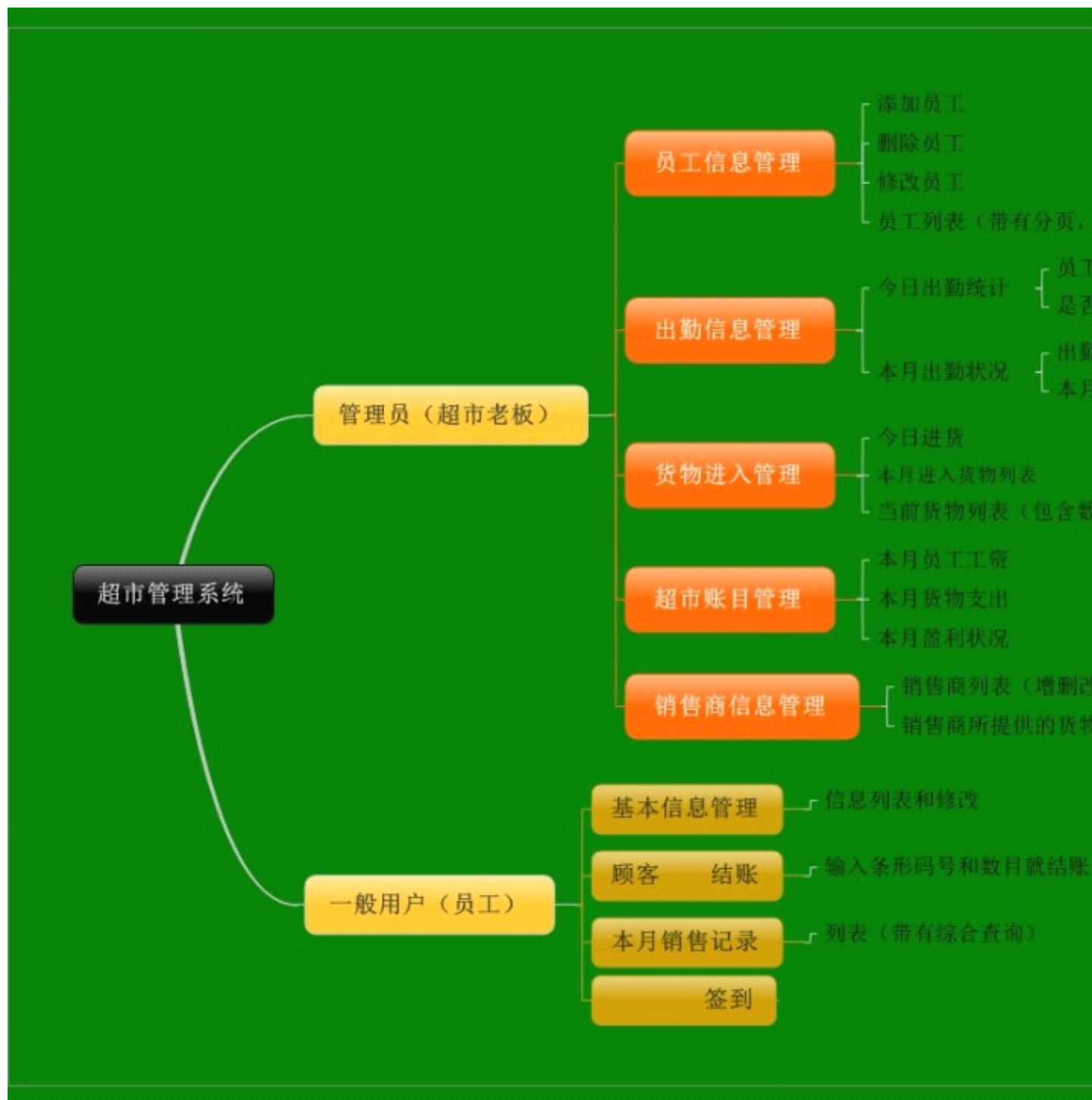
本系统的开发目标包括:

- 2 减少超市人力与管理费用;
- 2 为超市提供方便，快捷的结账体系;

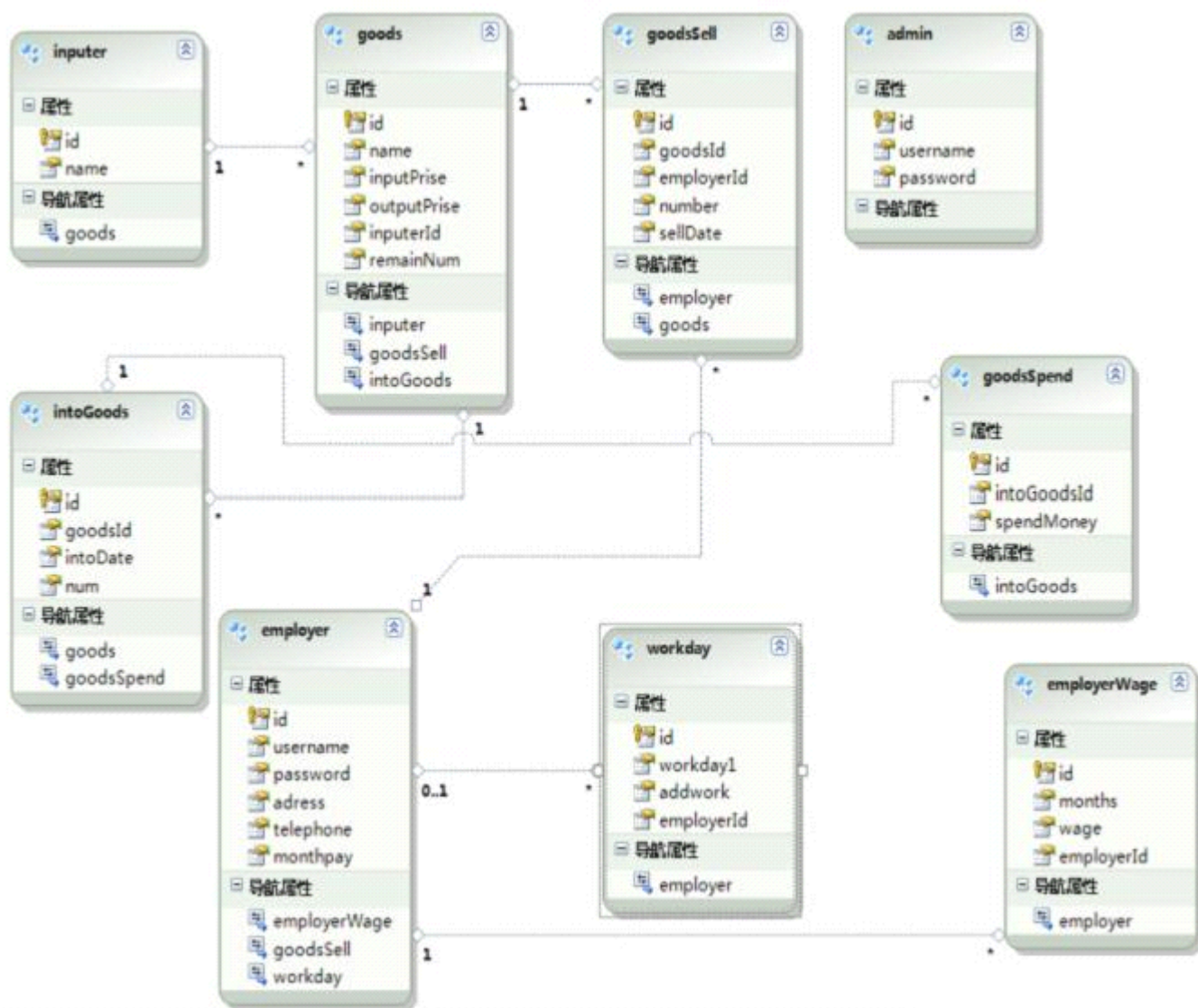
² 为超市提供准确，高效的库存和财务管理系统；

² 为超市管理人员提供强大的管理和统计商品，资金的功能。

主要功能如下：



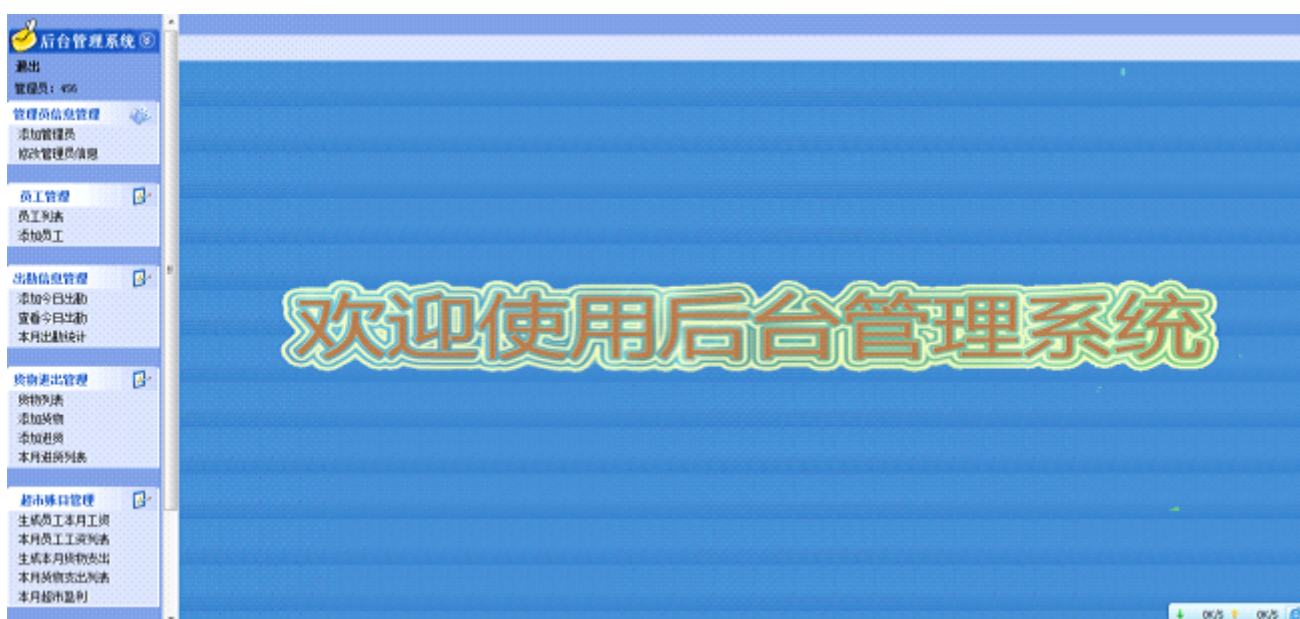
类的设计·



登陆:



登陆之后主页面:



员工列表界面：

查询条件

姓名: 家庭住址: 电话: 月薪: 从 至

查询

操作	编号	姓名	住址	电话	月薪
 	1	曹胜欢	山东济宁	123456	5000.0
 	2	曹宇	山东马坡	147258	3000.0
 	6	小金金1	山东德州	258585	8000.0
 	7	金真诚	山东济宁	123456	5000.0
 	8	杨过	山东马坡	147258	3000.0
 	9	小龙女	山东德州	258585	8000.0
 	10	张兴建	山东济宁	123456	5000.0
 	11	网通里	山东马坡	147258	3000.0
 	12	痴心	山东德州	258585	8000.0
 	13	孙老师	山东济宁	123456	5000.0

第1/3页 首页 1 2 3 尾页 第0页 Go

修改员工信息：

查询条件

姓名: 家庭住址: 电话: 月薪: 从 至

查询

操作



























电话

月薪

123456

5000.0

147258

3000.0

258585

8000.0

123456

5000.0

147258

3000.0

258585

8000.0

123456

5000.0

147258

3000.0

258585

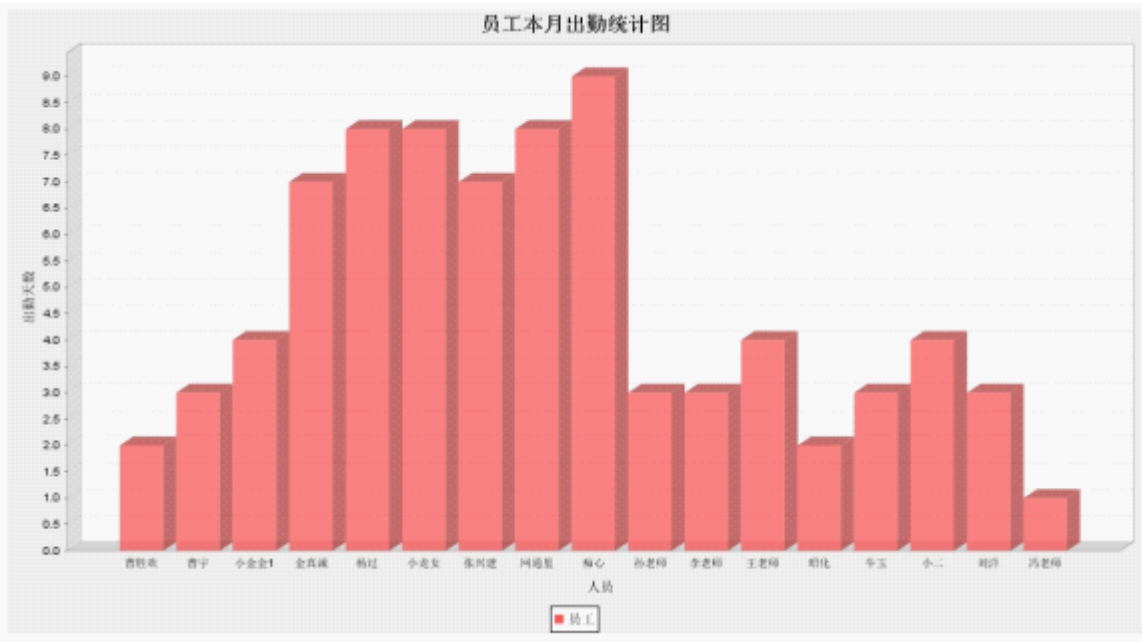
8000.0

123456

5000.0

第1/3页 首页 1 2 3 尾页 第0页 Go

员工出勤统计：



货物列表:

编号	商品名	进货时间	进货数量	进货厂商	花费金额
工资_12	火腿肠	2012-06-26 00:00:00.0	20	滨州学院	40.0
工资_13	面粉	2012-06-26 00:00:00.0	30	马甲小区	1500.0
工资_14	本子	2012-06-26 00:00:00.0	40	滨州学院	80.0
工资_15	书包	2012-06-26 00:00:00.0	50	马甲小区	500.0
工资_16	书包	2012-06-26 00:00:00.0	60	马甲小区	600.0
工资_17	香蕉	2012-06-26 00:00:00.0	70	滨州学院	350.0
工资_18	香蕉	2012-06-13 00:00:00.0	30	滨州学院	150.0
工资_19	面粉	2012-06-26 00:00:00.0	50	马甲小区	2500.0
工资_20	本子	2012-06-26 00:00:00.0	50	滨州学院	100.0
工资_21	面粉	2012-06-26 00:00:00.0	50	马甲小区	2500.0
工资_22	本子	2012-06-05 00:00:00.0	50	滨州学院	100.0
工资_23	书包	2012-06-05 00:00:00.0	50	马甲小区	500.0
工资_24	火腿肠	2012-06-06 00:00:00.0	50	滨州学院	100.0
工资_25	火腿肠	2012-06-04 00:00:00.0	80	滨州学院	160.0
工资_26	书包	2012-06-05 00:00:00.0	90	马甲小区	900.0

用力点击：[下载源码](#)

本程序由于时间仓促。一些原先设计好的功能没有能完全实现，本程序原本想加入普通员工的操作客户端，普通员工的签到、售货、根据售货多

少得到相应的奖金、本月销售记录等功能尚未完成，另外还可以扩展很多功能，由于时间仓促，所以暂时先实现这些

三：北京实习总结——记住牛人那些话

本文来自：曹胜欢博客专栏。转载请注明出处：

<http://blog.csdn.net/csh624366188>

短短的北京实习的日子，简单的可以概括为下面几个字：“痛并快乐着”。实习的地方是一个创业团队，他们主要是做自己的产品的研发，现在产品已经到了快接近尾声了，应该用不了多长时间就要上线了吧，我去了其实就算是一个打杂的，本来对产品又不熟悉，用到的技术.Net 也不是很熟。总体来说，这次实习的收获是正确的看到了自己的位置，也看到了自己的缺点，也为自己以后的就业和发展确定了一些方向。

这次实习机会确实也暴露出了自己很多的问题，比如：没有足够的观察力、对于问题的处理缺乏冷静思考等等诸多问题。这次机会主要要感谢一下这个开发团队的灵魂人物——勇哥，CSDN 中应该知道他的人很多，但真正了解他的人应该不算太多吧。其实我只想说他却是一个不折不扣的高手，这段时间我被彻底的臣服了。他是一个工作17年的老手了。他的思想给了我巨大的震撼，这次总结应该主要总结的是他对我说过的话，每一句话都需要我花很大的时间去揣摩，下面我就根据他对我说过的话来总结一下自

己在这次实习过程中发现的自己的不足：

1.学会模仿，去模仿牛人的代码比自创自己的代码风格要好的多。

在实习的这段日子里确确实实的让我感觉到了自己代码风格的缺陷，越来越发现自己的代码写的很烂，很难看，没有成熟的代码风格，想到哪写到哪，最后代码乱到连自己都看不懂了

2.善于观察，培养自己具有福尔摩斯式的思维能力。

这句话是牛人给我说的最多的一句话，原本说实话我只是听说过福尔摩斯，但不知道他是干什么的。最后看完大侦探福尔摩斯这部电影我才真正明白牛人想要告诉我的是什么，通过观察，很多问题没有我们想象的那么难，培养自己福尔摩斯式的思维有利于自己对问题的解决能力和分析能力的提高。不要遇到问题就立马想到百度或者 google 或者去问别人。要善于思考

3.不要轻易的去问别人问题。

在很多公司里，刚来的新人一般都会给配备一个老师。但是不要把这位老师当做你自己的依靠，遇到问题要先尝试着自己去解决，不要问一些没价值的问题，大家要知道，一个老程序员在公司的工作能力差不多能赶得上五个新程序的工作能力，所以，你打扰师傅半个小时，相当于接近自己三个小时的时间，尝试着去解决问题，千万切记，同一个问题不要重复问两遍。这也是师傅所忌讳的事情

4.英语的学习方法。

英语对一个程序员来说是非常重要的能力，为了应付考试式的英语

学习永远不会有大的提高，要学会去用他。在平时去学习他，没事多看一些英语的技术文档或者博客，这样既收获技术也收获了英语，一举两得。

5.不要为了想学什么而去学习，要去为了用他而学习。

有时候我们会突然发现我们还有一个知识点没有学习呢，于是就想去学习这个东西，这时候我们可能不知道，这样的学习效率是非常低的。我们只是想学他，没有真正的知道他该怎么用，或者说没有去用，这样就算学会了，过不了几天也就忘了。为了用它而去学习，这样的好处是，我们不用专门抽出时间来学习某一个知识点，在我们尝试用的过程中我们就把它学会了。这样大大的加深了我们的印象，一般也不是那么容易就忘掉的。

6.不要轻易说不：

如果在工作过程中，老板交给你一个任务，千万不要轻易说不，不要说，我没做过，我不会做。这样的话只能说明你这个人能力有问题。这也是老板最不想听到的话，不要轻易说不。我们不会，完全可以借这个机会去学习，去查资料，但我们不能说不。

7.不要问我该怎么做，而去问我该做什么。

这句话应该和上边的不要轻易说不差不多。不要轻易的去问别人这个问题我该怎么做，我没有一点思路，等着别人给你说答案，说实话，这种行为时可耻的。我们应该最多也就是问一句我该做什么，你说做什么，剩下的我来去做，不要让别人喂你东西吃。

总结起来，虽然实习这段时间没有真正做过什么太有技术含量的工作，但是在思想和编码规范上却是还是学到了很多，这也为以后更好地去学习打下了一定的基础。从老板给我说的这些话中我们也可以清楚的看到，

不要一味的去追求新技术，要去完善自己的各种能力，学习能力，观察能力等等，思想要跟的上“潮流”。